

Concurrent Programming in Dao

Limin Fu (phoolimin@gmail.com)

September 19, 2009

1 With Coroutine

Dao supports coroutine (*collaborative multithreading*) in a similar way as Lua. In Dao, a coroutine object is actually a virtual machine process with its own calling stack of contexts. The execution of a coroutine object is resumed by **coroutine.resume()**, and suspended by **coroutine.yield()**. Please see *Lua Manual* for more information about coroutine.

Though the Dao coroutine APIs are almost identical to that of Lua, there are minor differences which are documented here *Dao Coroutine API*.

Here is a simple example adapted from Lua Manual,

```
routine foo( a )
{
    stdout.println( "foo: ", a );
    return coroutine.yield( 2 * a );
}

routine bar( a, b )
{
    stdout.println( "co-body: ", a, "\t", b );
    r = foo( a + 1 );
    s;
    { r, s } = coroutine.yield( a+b, a-b );
# raise Exception.Error( "raised" );
    stdout.println( "co-body: ", r, "\t", s );
    return b, "end";
}

co = coroutine.create( bar, 1, 10 )

stdout.println( "main", coroutine.resume( co ) )
stdout.println( "main", coroutine.resume( co, "r" ) )
stdout.println( "main", coroutine.resume( co, "x", "y" ) )
stdout.println( "main", coroutine.resume( co, "x", "y" ) )
```

2 With Native Threads

Dao Language has built-in supporting for multi-threaded programming. The threading facilities are accessible through the multi-threading library named *mtlib*, which can be used to create thread, mutex, condition variable and semaphore etc, and can be used to perform other threading operations.

2.1 Thread Creation

It is extremely simple to create a thread in Dao, just like this,

```
thread_id = mtlib.thread( function / object.method, p1, p2, ... )
thread_id = mtlib.thread( function_curry )
```

Note that if *function* or *object.method* have overloaded versions, *mtlib.thread()* is able to invoke the corrected one based the parameters supplied after the first paramter.

```
routine MyRout( name : string )
{
    stdio.println( "string parameter = ", name )
}
routine MyRout( index : int )
{
    stdio.println( "int parameter = ", index )
}

class MyType
{
    public

    routine show( name : string ){ stdio.println( "show string: ", name ); }
    routine show( index : int ){ stdio.println( "show number: ", index ); }
}

t1 = mtlib.thread( MyRout, "DAO" )
# or: t1 = mtlib.thread( MyRout{"DAO" })
t2 = mtlib.thread( MyRout, 111 )
# or: t2 = mtlib.thread( MyRout{111 })
t1.join();
t2.join();

mt = MyType();
t1 = mtlib.thread( mt.show{ "DAO" } )
t2 = mtlib.thread( mt.show{ 111 } )
t1.join();
t2.join();
```

2.2 Synchronization

If a program runs multiple threads concurrently without synchronization, for different execution it may execute threads in different way and give different results. To make a multi-threaded program give predictable result, one can use mutex, condition variable and semaphore to synchronize the threads in a desired manner.

2.2.1 Mutex

Mutex can be used to synchronize the accessing of shared data structures. It has two state: locked and unlocked. A mutex can be locked by only one thread. A thread is suspended when it attempt to lock a mutex which has been locked by another thread. Mutex can be created by the **mtlib** library object,

```
mutex = mtlb.mutex();
```

Then it can be locked or unlocked by,

```
mutex.lock();  
mutex.unlock();  
mutex.trylock();
```

By calling *lock()* , the calling thread will be block if the mutex is already locked by another thread. If the mutex is locked by the same thread, a second calling of *lock()* may cause a deadlock. *trylock()* is the same as *lock()* except that it will return immediately instead of blocking the calling thread if the mutex is already locked.

2.2.2 Condition Variable

Condition variable is a synchronization device which allow a thread to be suspended if a condition is not satisfied and resume execution when it is signaled. The basic operations on a condition is: wait on the condition, or signal the condition.

```
condvar = mtlb.condition();
```

Condition variable should always be used together with a mutex. To wait on a condition,

```
mtx.lock()
```

```
condvar.wait( mtx );  
mtx.unlock();
```

To wait on a condition for a maximum time,

```
mtx.lock()  
condvar.timedwait( mtx, seconds );  
mtx.unlock();
```

seconds can be a decimal, for example, `condvar.timedwait(mtx, 0.005)` will wait for five milliseconds.

2.2.3 Semaphore

Semaphore can be used to set a limit on resources. It maintains a count for the resource, and allows a thread to proceed, when it attempts to decrease a non-zero count. If the count already reaches 0 before the decrement, the thread will be suspended until the count becomes non-zero. When the thread finished using the resource, it should increase the count of semaphore. A semaphore must be created with an initial count,

```
sema = mtlib.semaphore( count );
```

To access a resource guarded by a semaphore, use,

```
sema.wait()
```

If the resource is acquired, the count of *sema* will be decreased.

To release the resource, use

```
sema.post()
```

which will increase the count.

2.3 Cancellation

In Dao, all threads are created with cancellable state. A thread *thread_id* can be cancel by other thread by

```
thread_id.cancel()
```

In a thread, only certain function calls create cancellation points. But one can explicitly insert cancellation point by using

```
mtlib.testcancel()
```

in some points to ensure a thread really cancellable.

There is no need for user defined clean up after cancellation or finishing a thread, the interpreter will take care of them all, including garbage collection (obviously) and even mutex unlocking of mutex locked by the thread.

2.4 Thread Specific Data

In Dao, it is also extremely easy to setup and use thread specific data. The library object *mtlib* or thread id object has a method named *mydata()* , which returns a thread specific hash, so,

```
mtlib.mydata()[ key ] = data;  
mtlib.self().mydata()[ key ] = data;
```

will create *data* specific to the current thread and accessible with key *key* , throughout the thread, since *thread* is global.

```
routine MyRout( name : string )  
{  
    stdio.println( "string parameter = ", name )  
    mtlib.mydata()["data"] = name;  
}  
  
t1 = mtlib.thread( MyRout, "DAO" )  
t2 = mtlib.thread( MyRout, "LANGUAGE" )  
t1.join();  
t2.join();  
  
stdio.println( t1.mydata()["data"] );  
stdio.println( t2.mydata()["data"] );
```

3 3. With Asynchronous Function Calls

Probably, the simplest way to create multi-threaded programs in Dao is to use asynchronous function calls (AFC). The way of using AFC is almost identical to that of normal function

calls, with the exception that the keyword **async** must follow the call.

```
myfunc( myparams ) async;  
myobj.mymeth( myparams ) async;
```

Any functions or methods can be invoked in such asynchronous way!

Normally AFC is executed in a separated native thread, which can be either an idle thread available from the thread pool, or a new thread created on the fly. There is a limit on the size of the thread pool. If this limit is reached, and there is no idle thread available from the pool, the AFCs are scheduled to run when some threads become idle.

Though any functions or methods can be invoked asynchronously, this does not mean they can really be running concurrently in the same time. If multiple AFCs are invoked for the methods of the same class instance, the Dao VM will schedule them to be run sequentially, to guarantee that the members of the class instance will be accessed and/or modified sequentially. In fact, the AFC mechanism is implemented on the top of the Actor Model (see below). In this case, the functions, class instances, C objects and virtual machine process etc. are the actors. As an intrinsic synchronization mechanism, one principle of the Actor Model is that an actor must process/respond sequentially (to) the messages sent to itself.

However, the Dao VM does not have internal synchronization for data shared in other ways such as parameter passing or global variables etc. For such shared data, mutex, condition variable and semaphore can be used for synchronization. If the current thread has to wait for the completion of all the AFCs invoked by itself before preceeding, keyword **join** can be used together with keyword **async** in the last AFC to join all the AFCs with the current thread.

There is another keyword that can also be used with **async** , that is, **hurry** . Note that, although the garbage collector (GC) of the Dao VM runs concurrently with the program threads, there is a limit on the number of garbage candidates that can be handled by the collector. Once this limit is reached, the program threads are blocked until the collector finishes processing garbages, with the exception of the main thread that is never blocked. If an AFC is used to handle "urgent" task, **hurry** can be used to indicate that the thread running this AFC should not be blocked by the GC.

The return value of AFC is a future value, which is a class instance of a Dao class named **FutureValue** . When the AFC is finished, the **Value** field of this future class instance is set with the returned values of the AFC.

Example,

```
routine test( name : string )  
{
```

```

    i = 0;
    while( i < 5 ){
        stdio.println( name, " : ", i );
        stdlib.sleep( 1 );
        i ++ ;
    }
    return "future_value_"+name;
}

a = test( "AAA" ) async;
b = test( "BBB" ) async;

list = {};
list.append( test( "CCC" ) async join );

stdio.println( a );
stdio.println( b );
stdio.println( list );

stdio.println( a.Value );
stdio.println( b.Value );
stdio.println( list[0].Value );

```

There following is two simple implementations of parallel merge sort algorithm:

```

routine merge( list1, list2 )
{
    list = {};
    N = list1.size();
    M = list2.size();
    i = 0;
    j = 0;
    while( i < N && j < M ){
        if( list1[i] < list2[j] ){
            list.append( list1[i] )
            i ++;
        }else{
            list.append( list2[j] )
            j ++;
        }
    }
    if( i < N ) list += list1[ i : ];
    if( j < M ) list += list2[ j : ];
    return list;
}

routine merge_front( list1, list2, front, back )
{
    N = list1.size();
    M = list2.size();
    S = N+M;
    i = 0;
    j = 0;
    while( i < N && j < M && front.size()+back.size()<S ){

```

```

        if( list1[i] < list2[j] ){
            front.append( list1[i] )
            i ++;
        }else{
            front.append( list2[j] )
            j ++;
        }
    }
}
routine merge_back( list1, list2, front, back )
{
    N = list1.size();
    M = list2.size();
    S = N+M;
    i = N-1;
    j = M-1;
    while( i >=0 && j >=0 && front.size()+back.size()<S ){
        if( list1[i] > list2[j] ){
            back.pushfront( list1[i] )
            i --;
        }else{
            back.pushfront( list2[j] )
            j --;
        }
    }
}

routine sort( list )
{
    N = list.size()
    if( N <= 1 ) return list;

    sublist1 = sort( list[ : N/2-1 ] ) async;
    sublist2 = sort( list[ N/2 : ] ) async join;

    # return merge( sublist1.Value, sublist2.Value );
    front = {};
    back = {}
    merge_front( sublist1.Value, sublist2.Value, front, back ) async;
    merge_back( sublist1.Value, sublist2.Value, front, back ) async join;
    return front + back[ front.size()+back.size()-N : ];
}

list = { 213,1,35,27,49,55,63,75,87,99,115,103 };
sorted = sort( list );

stdio.println( sorted );

```

4 4. With Message Passing Mechanism Based The Actor Model

(experimental)

With Message Passing Interface APIs provided in library *MPI*—<http://xdao.org/weblet.dao?WIKI>, one can easily do concurrent and distributed programming in Dao. In the MPI library, there are 3 principle functions: `spawn()`, `send()` and `receive()`. With `spawn()`, one can create lightweighted virtual machine processes or real operation system processes, in the local or remote computers; and with `send()` and `receive()`, a process can send message to or receive message from other process.

The Dao MPI uses network sockets to do all the inter- OS process and inter-computer communications. When a Dao program is started with a process name specified with `“-pNAME”` or `“-proc-name=NAME”` in the command shell, it will bind to a port for accepting communications from other processes. By default, it will try to bind to port number 4115 (D:4, A:1, O:15), if this port is already used, it will try other port to bind. In each computer, the process that binds to port 4115 is the master process, which must be running if Dao program from other computer need to spawn a VM/OS process on this computer.

Each named OS process running Dao program is identified by the host name and the port it binds to. VM processes within an OS process are identified by names. In general, to be usable by the Dao MPI, the process name (actor address) has the form of `“vm_proc@os_proc@@host”` (`“os_proc”` is mapped to the port number), which is called `“pid”` (process identifier) for simplicity. A pid does not need to have the full form, some parts can be omitted. See also *MPI*—<http://xdao.org/weblet.dao?WIKI> . The following figure displays the relationships between process at different scope.

Each VM process will be running or schedule to run in a separated thread drawing from a thread pool or created on the fly, and return the thread to the pool when `mpi.receive()` is called. Each OS thread may run different VM processes at different time.

Message Passing Interfaces are available in library **mpi** .

4.0.1 Spawn Process

The prototype of `mpi.spawn()` is

```
mpi.spawn( pid :string, proc :string, ... )
```

If `pid` is of form `“”`, `“vm_proc”`, `“vm_proc@os_proc”`, `“vm_proc@os_proc@@host”`, a VM process will be spawned, and `proc` should be the name of the routine to be spawned.

The rest of the parameters will be passed to the routine when the process is created (currently passing additional parameter is not supported for pid form "*vm_proc@os_proc*", "*vm_proc@os_proc@@host*"). If the pid is an empty string, then spawned VM process has no pid, and must be accessed by the process handle returned by **mpi.spawn()** .

If *pid* is of form "*@os_proc*", "*@os_proc@@host*" , an OS process will be spawn in the current computer or a host in the network, and *proc* should be the name of the Dao script file to be executed. The rest parameter is the timeout for spawning. If the OS process does not spawn successfully within the timeout, an exception will be raised. The default value of the timeout is -1. No positive timeout means infinite waiting.

4.0.2 Send Message

The prototype of **mpi.send()** is

```
mpi.send( object, ... )
```

If *object* is a process identifier, send the rest of parameters as message to it; If *object* is a callable object, this object is scheduled to be called asynchronously with the rest parameters. Note, message is sent out asynchronously, and only number, string, complex number and numeric array can sent through this interface.

If the process identified by *object* , can not be found, an exception will be raised. But if the process is already dead, no exception will be raised, it is up to the user to implement such checking.

4.0.3 Receive Message

The prototype of **mpi.receive()** is

```
mpi.receive( timeout=-1 )  
mpi.receive( pid :string, timeout=-1 )
```

If **mpi.receive()** is called without parameters, the calling process will wait for messages from any process, and will be paused definitely until a message arrives. The received message including the pid of the sender will be packed into a list and returned by this API. To avoid it waiting for infinite long time, a timeout can be passed as parameter to **mpi.receive()** , in this case, if no message arrive within the timeout, **mpi.receive()** will return with an empty list.

If one want to receive message from a specific process, the pid of that process can be passed to **mpi.receive()** as the first parameter. Messages from other process will be stored in

the "mailbox", until **mpi.receive()** has received and processed a message from that specific process. A timeout can also be specified as the second parameter. By passing the pid of the expected process to **mpi.receive()**, synchronous communication can be realized.

4.0.4 Example

File *mpi_spawn.dao*,

```
stdio.println( "=====" );
mpi.spawn( "@pid", "mpi_script.dao" );
mpi.spawn( "vmp@pid", "test" );

mpi.send( "@pid", "TO MAIN" );
mpi.send( "@pid", "TO MAIN" );

stdio.println( mpi.receive() );

stdio.println( "=====" );
mpi.spawn( "@pid1", "mpi_script.dao" );
mpi.spawn( "vmp@pid", "test" );
mpi.send( "vmp@pid", "MESSAGE" );

mpi.spawn( "@pid2@@localhost", "mpi_script.dao" );
#mpi.spawn( "@@pid2@pid", "mpi_script.dao" );
stdio.println( "here!!!!!!!!!!!!!!!!!!!!!!!!!!!!" );
mpi.spawn( "vmp@pid2@@localhost", "test" );
mpi.send( "vmp@pid2@@localhost", "ANOTHER", 123.456 );
```

File *mpi_script.dao*,

```
stdio.println( "mpi_script.dao spawned" );

routine test()
{
    stdio.println( "start test" );
    i = 0;
    while( i < 10 ){
        msg = mpi.receive();
        stdio.println( "test() ",msg );
        mpi.send( msg[0], "FROM test()" );
        i ++;
    }
}

msg = mpi.receive();
stdio.println( msg );
mpi.send( msg[0], "CONFIRMED" );
mpi.send( "main", "message from main" );
mpi.send( "main", "message from main" );
```

```
stdio.println( mpi.receive() );
stdio.println( mpi.receive() );
stdio.println( mpi.receive() );
stdio.println( "xxxxxxxxxxxxxxxxxxx" );

while( 1 ){ stdlib.sleep(10) }
```

File *mpi_send.dao* ,

```
mpi.send( "vmp@pid", "FROM ANOTHER OS PROCESS" );
stdio.println( mpi.receive() );
```

Run *dao -pmaster mpi_spawn.dao* first, then run *dao -ptest mpi_send.dao* .