Concurrent programming with Concurnas

Introducing the Concurnas language

Concurrency in Concurnas

Reactive computing with Concurnas

Under the hood of concurrency

Graphics processing units

Multiparadigm programming

Working with data

Installing Concurnas

Conclusion

CODING

Concurrent programming with Concurnas

Modern developers have access to hardware of never-before-seen power. Here's how to leverage Concurnas, a new JVM programming language, to unlock that power.

by Jason Tatton

June 22, 2020

Writing concurrent programs is hard. Yet to leverage the power of today's modern multicore CPU architectures, software written for concurrency is a necessity. Furthermore, most computers now have access to graphics processing units (GPUs) that give developers the opportunity to quickly solve massively parallel problems at unprecedented speed. But native coding for GPUs is also hard.

The way in which developers build software has changed. More and more systems are event-driven or reactive in nature, but most programming languages and algorithms do not model well in this new paradigm. These challenges come at a time when developers are expected to be more productive and to produce reliable, performant software that scales. The time has come for innovative solutions to these problems.

This article explores Concurnas, a new programming language targeting the JVM. Concurnas was especially designed for building modern, reliable, scalable, high-performance, concurrent, distributed, and parallel systems.

Introducing the Concurnas language

Concurnas was designed with four core objectives:

- To make it easier—and more accessible—for the average developer to build concurrent and parallel systems that take advantage of today's multicore CPU and GPU hardware architectures

- To be an easy-to-learn and performant language by providing the syntax of a dynamically typed language with the performance and functionality associated with a strong static type system
- To be of utility to seasoned software engineers as well as to specialists in fields other than computing
- To deliver multiparadigm language support by providing the most popular features of the imperative, object-oriented, functional, and reactive programming paradigms in a concise language that supports both prototyping and software at scale

Concurnas compiles for execution on the JVM, and in doing so, it gains the incredible performance of the JVM, automatic memory management (aka garbage collection), just-in-time compilation, access to the Java standard library (referenced in the examples below), tools, and compatibility with existing JVM languages including Java, Scala, and Kotlin.

The Concurnas language's core syntax was inspired by Python. The following is a simple function for calculating the greatest common denominator of a number and the value `25`:

```
def gcd(x int){
  y = 25
  while(y){
    (x, y) = (y, x mod y)
  }
  x
}
```

Here are some noteworthy items in that code:

- The type of the variable `y` is inferred to be of type `integer` as is the return type of the overall function. With the exception of function input parameters, Concurnas is an optionally typed language that relies upon type inference (a feature prevalent in functional programming languages) to present the facade of a dynamically typed language for ease of use.
- The `return` statement is omitted from the definition of the function. All code blocks, which are denoted with `{}`, are able to return values. The last referenced value (`x`, in this example) is implicitly returned.
- Concurnas supports tuples. You can see the definition of a tuple, `(y, x mod y)`, and on the left side of the assignment operator, you can see its decomposition, `(x, y)`. Tuples are a nice means by which you can work with collections of data without having to define wrapper classes to collate them together.
- All types may be used where a boolean expression is expected, such as within `while(y)`.
- Functions may exist in isolation; they are not obliged to be defined within the context of a class as they are in languages such as Java.

Concurrency in Concurnas

Concurnas presents an innovative alternative to the standard model of concurrency found in Java and many other JVM languages. The Concurnas model of concurrency avoids the need to explicitly manage threads, critical sections, locks, and shared memory.

This is achieved by performing computation in lightweight thread-like containers known as *isolates*. All code in Concurnas programs runs in isolates. Isolates are able to execute concurrently and cannot directly share memory.

If data is referenced across different isolates, it is copied, thus preventing accidental sharing of state. Of course, if this were the end of the story, this would be a programming language that would not be very functional, because controlled sharing of state, in a safe manner, is necessary for concurrent computing.

To this end, Concurnas provides a special type known as a *ref*. A ref may be composed of any type, and it is denoted by appending a colon (`:`) to a type. Since refs manage their own concurrency in a safely atomic manner, they are shared between isolates without copying. If a return value from an isolate is expected, the return value will be of ref type. Here is how to create some isolates making use of the previously defined `gcd` function:

```
res1 int: = gcd(92312)!
res2      = {gcd(2438210)}!

larger = res1 if res1 > res2 else res2
```

Above, I create two isolates using the exclamation point, or bang, operator (`!`). The `!` operator may be appended to function invocations or blocks of code.

The return type of both isolates is `int:`. I chose to explicitly define `res1` as being of type `int:` but I omitted the explicit definition for `res2`. Subsequently, the code takes the two concurrently calculated results held in `res1` and `res2` and determines which one is larger. When the final statement of the program is reached, the code will pause execution until both `res1` and `res2` are set.

While execution is paused, the underlying isolate executing the code will yield for other isolates to execute. When `res1` and `res2` have been set, control will then be transferred back to the isolate as it dereferences the contingent refs and uses their values for subsequent execution of the code.

Use the `await` keyword, with an optional guard condition, to pause isolate execution contingent upon the state of one or many refs being set, for example:

```
result int:

{result = 100}!

await(result)
//now execution may continue...
```

Another alternative is to use the `sync` keyword, which ensures that all isolates created within a code block have completed execution before carrying on with execution:

```
result int:
sync{
    {result = 100}!
    //...more isolates defined here...
}

assert result == 100
```

The beauty of the isolate model is that developers can write code in a largely single-threaded manner, without having to manage concurrency control. When it comes to running that code in a concurrent setting, spawning isolates provides the reassurance that the code will behave in the same way as if it were executed in a single-threaded setting. This model separates concerns within the code.

A big part of this is in the implicit copying of dependent data between isolates. Here is an example:

```
n = 10

nplusone  = { n += 1; n }!
nminusone = { n -= 1; n }!

assert n == 10
assert nplusone == 11
assert nminusone == 9
```

The code above spawns two isolates, both of which operate upon a variable `n`. Since isolate execution is intrinsically nondeterministic, perhaps the two isolates are scheduled to run at exactly the same time; maybe the first one executes before the second, or perhaps the second executes before the first. It's unknown, but that doesn't matter.

Because of dependent data copying, no matter how the underlying isolates are executed, the same results are always derived and the *source*, `n`, is unchanged. This implicit copying of data also means that you do not need to introduce complex immutable data structures. Instead, you may use the mutable data structures that you are familiar with from introductory

college computer science and that are prominent in most real-world software.

There are times where applications need to work with data structures, which might be large—maybe many megabytes. Those data structures need to be shared across concurrent entities. In these sorts of situations, copying data would have a detrimental effect upon system performance. For that, Concurnas provides the `shared` keyword. This keyword permits isolates to violate the condition of "no shared state," and it is a great option to use with large read-only data structures and also when you reference classes from other JVM languages that are known to be thread-safe and, therefore, appropriate to run in a concurrent setting. Here's an example:

```
shared lotsOfNumbers = x for x in 0 to 100 st
//shared suppresses copying of lotsOfNumbers

def filteredDataSet(alist java.util.List<int>
    x for x in alist if x mod modof == 0
    //above is an example of list comprehensi
}

mod15 = filteredDataSet(lotsOfNumbers, 15)!
//mod15 ==> [0, 15, 30, 45, 60, 75, 90]
mod16 = filteredDataSet(lotsOfNumbers, 16)!
//mod16 ==> [0, 48, 96]
```

An alternative approach to the sharing of data in the above manner is to use an *actor*. Actors are a great way of producing microservices within your code. Actors look just like regular classes in object-oriented languages such as Java. They export a simple isolate-like model of computation whereby all execution within an isolate is performed in a single-threaded manner. This is achieved by implicitly spawning a dedicated isolate within the actor, whose job it is to specifically execute calls made to that actor. Here is an actor:

```
actor IdGenerator(prefix String){
    //IdGenerator has one constructor taking
    cnt = 0//implicit private state
    def getNextId(){
        toReturn = prefix + "-" + cnt
        cnt += 1
        toReturn
    }
}

//let's use it:
idGen = IdGenerator("IDX")//create an actor
anId1 = idGen.getNextId()//==> IDX-0
anId2 = idGen.getNextId()//==> IDX-1
```

Above, because calls to `IdGenerator`'s `getNextId` method are within an actor, multiple isolates may call it without there ever being scope for inconstancy of state within the

`IdGenerator`, so there's no potential for two IDs of the same value to be returned to multiple isolates. In Java, you would need to explicitly introduce a critical section around `getNextId` via locks or the `synchronized` keyword.

Actors may also be created of existing classes. Here's how to create an actor service out of a `HashSet`:

```
setService = actor java.util.HashSet<int>()
setService.add(65)
```

The number of isolates that can be created is limited only by the amount of memory available on the physical machine running the code. This limit is much higher than the number of threads that can be spawned in a JVM. In this way, the isolate model of concurrency scales much better than conventional approaches. This also means that for library writers who want to take advantage of concurrency in their solutions, Concurnas allows parallelism without the need to create additional threads or require threads to be injected into the libraries by the host program.

Reactive computing with Concurnas

A unique property of refs in Concurnas is the ability for isolates to register interest in changes of their state. It is through this means that developers are afforded the capabilities of reactive computing.

Imagine you are building a finance application that initiates a single trade on the stock market when the relative price of an asset exceeds that of another asset. The application needs to consume two feeds of market data, perform some calculation upon their latest state, and potentially initiate some action contingent on the results of that calculation. This can be implemented using the `every` keyword:

```
asset1price int:;
asset2price int:;

every(asset1price, asset2price){
    if(asset1price > asset2price){
        //... initiate trading action here!
        return//terminate future invocation o
    }
}
```

The `every` block above will trigger every time `asset1price` or `asset2price` is updated. Reactive components may themselves return refs holding streams of returned values. As such, it's possible to chain together reactive components, for example:

```
a int:
b int:

c = every(a, b){
    a + b
}

every(c){
    System.out.println("latest sum: {c}")
}
```

Above, `c` will equate to the sum of `a` and `b` each time `a` or `b` changes. The code has another `every` block that is responsible for printing the latest value of `c`.

Concurnas presents a natural way of expressing these sorts of reactive systems, but it also aims to be a compact language that is both easy to write and read. So, a more compact version of the above syntax is this:

```
c <= a + b
```

As before, `c` will equate to the sum of `a` and `b` each time either `a` or `b` changes.

Under the hood of concurrency

The concurrency model in Concurnas is implemented using continuation passing. This is very similar to the approach taken to implement continuations in Project Loom. At runtime, isolates are mapped to a fixed number of underlying hardware threads, the number spawned being contingent upon the compute capability of the underlying hardware available for execution, such as the number of CPU cores available.

When an isolate arrives at a point where it's necessary to pause the execution of code—such as when a ref is dereferenced, but no value has yet been assigned to that ref—the current execution stack of the program is packaged up the call chain until execution arrives at the underlying worker thread instance executing that isolate. At this point, execution transitions to a different isolate if one is available; otherwise, execution waits until it has been notified that it may resume execution of the paused isolate, for example, by a value being set for the ref upon which execution was paused.

This approach has a small performance penalty. In practice, for an application that involves heavy switching between isolates, the penalty turns out to be around 5% additional runtime. I think this is a reasonably small price to pay for such transformative functionality.

The current mechanism by which isolates are scheduled is round robin–based, although as the Concurnas language and runtime platform evolves, it might leverage the work-stealing scheduler exposed in Java Fork/Join as well as allow end users to define their own schedulers, divvying up resources as they see fit. This scheduling mechanism is akin to selecting or tweaking a garbage collector, something that most developers won't need to worry about. Of course, a few developers will find this capability tremendously useful.

This process of stack unwinding, copying state between isolates, and rendering code to make it suitable for concurrent execution is performed at runtime within the Concurnas class loader. This provides compatibility with code written in other JVM languages: Concurnas code compiled today will be compatible with later versions of Concurnas in the future, while incremental performance improvements continue to be added to the platform.

Graphics processing units

Most modern computing platforms have access to GPUs. A GPU is a massively data-parallel computation chip that is perfect for solving SIMD (single instruction, multiple data) operations and problems that are normally CPU-bound.

When compared to CPU-based algorithm implementations, a GPU might be able to solve a SIMD problem 100 times faster—maybe more. What's more, this is with a far-reduced energy and hardware cost per gigaflop relative to CPU-based computation. Looking at the specifications of modern CPUs and GPUs, it's easy to see why.

A top-of-the-line graphics card, for example, an NVIDIA GeFORCE RTX 2080 Ti, has 4,352 cores that can perform computation in parallel. Compare that to a state-of-the-art CPU, which might have 64 cores. Even though GPU cores are clocked at a much lower rate than CPU cores, suitable computing tasks have access to orders of magnitude more compute power with a GPU. Many of the world's leading supercomputers achieve their parallelization through the use of dedicated GPU hardware.

Now the bad news: Programming GPUs can be hard. Not only are GPUs optimized for solving only a certain (thankfully, quite broad) class of SIMD problems, but developers have often needed to have a deep understanding of the underlying hardware and to program that hardware using low-level languages such as C.

Concurnas builds in support for programming GPUs and associated parallel computing constructs. Idiomatic Concurnas code can be written to leverage GPUs and perform parallel computation in an efficient manner. This greatly reduces the barrier to entry for leveraging GPU computing. Code is executed

upon GPUs within kernels. Kernels are executed in parallel by the individual cores of the GPU.

Let's look at a simple example of a kernel now: to calculate `y = a**2 + b + 10` for each element of two arrays, `A` and `B`:

```
gpukernel 1 twoArrayOp(global in A float[], g
    idx = get_global_id(0)
    result[idx] = A[idx]**2 + B[idx] + 10
}
```

The above kernel operates on three arrays that are expected to exist in the global memory space of the GPU (analogous to RAM in terms of a CPU architecture). The code specifies whether the arrays the kernel operates upon are input or output arrays (mixed mode is also available), which allows the compiler to optimize the generated machine code for the GPU. Note that kernels cannot return values; rather, they are able to write their results to output variables. The call to `get_global_id(0)` provides context to the calling core with regard to which element of the array it should be operating upon. The code needs to identify and copy data to a GPU:

```
//select a GPU device...
device = gpus.GPU().getGPUDevices()[0].device

//create three arrays of size 10 on this GPU,
inGPU1 = device.makeOffHeapArrayIn(float[].cl
inGPU2 = device.makeOffHeapArrayIn(float[].cl
//and one as output
result = device.makeOffHeapArrayOut(float[].c

//now write to the arrays on the GPU
c1 := inGPU1.writeToBuffer([ 1.f 2 3 4 5 6 7
c2 := inGPU2.writeToBuffer([ 1.f 2 1 2 3 1 2
```

The `writeToBuffer` copy operation returns a ref (`c1` and `c2`). Because the operation is asynchronous, the code can carry out other operations concurrently while this takes place. Subsequent GPU operations may take these refs as arguments to build up pipelines of operations on the GPU. In a high-performance computing environment, where throughput is a concern, it is best to concurrently transfer data to and from a GPU while it is processing (other) data.

Concurnas models GPUs as coprocessors, so operations upon them take place asynchronously and concurrently to other isolates executing within the system.

Next, create a handle to the `twoArrayOp` kernel:

```
inst = twoArrayOp(inGPU1, inGPU2, result)
compute := device.exe(inst, [10], c1, c2)//ru
ret = result.readFromBuffer(compute)
```

After the `inst` handle has been passed to the `twoArrayOp` kernel with bounded input and output variables, it is passed to a device for execution via the `exe` method. The code also specifies the number of cores on the GPU to run and any refs for which to wait for completion before execution may take place. In this case, that means the refs `c1` and `c2`, corresponding to the operations to copy data to the GPU. Because this execution is an asynchronous operation, it too returns a ref: `compute`. Finally, read the resultant calculations from the results buffer, `result`, contingent upon `compute` having completed.

Depending on the hardware model, GPUs may have different memory spaces available to them, both global and local. When you develop an algorithm that has a large degree of spatial locality (such as matrix multiplication), using local memory can result in orders of magnitude of performance improvements over using global memory. Concurnas also provides support for the different memory spaces of GPUs.

Although an understanding of some of the nuances of GPU computing is a requirement to extract the best performance from a GPU, using Concurnas greatly simplifies this process. All this functionality is achieved by transpiling Concurnas GPU kernel code into OpenCL C. (*Transpiling* means taking code written in one language and transforming it into another language.) OpenCL is a multiplatform API for GPU computing supported by the three big GPU manufacturers: AMD, Intel, and NVIDIA. Transpiled code is then attached to compiled class files via an annotation for passing over to the GPU at runtime. This mechanism provides portability.

Multiparadigm programming

Concurnas is a multiparadigm language, which means it can cherry-pick the best and most popular aspects of the imperative, functional, object-oriented, and reactive programming paradigms.

When it comes to object-oriented programming, Concurnas presents a compact mechanism for defining classes. Many data-oriented classes are expressible in a single line of code, for example:

```
abstract class UniversityMember(~enrolled boo
class Person(~firstname String, ~surname Stri

//now create some instance objects:
p1 = new Person("dave", "brown", 1970)
p2 = Person("sandy", "smith", 1978, true)//us
```

Placing field definitions in parentheses after a class name indicates to Concurnas that you wish to automatically generate a constructor that can set those fields. Note that the `enrolled`

field has a default value of `false` (also, its type is inferred to be `Boolean`), and it can be optionally specified when creating an instance of the `Person` class. In the above example, `Person` inherits from `UniversityMember`, as signified through the use of the `extends` or `<` keyword. `Person` satisfies the superconstructor of `UniversityMember` by passing one of its input parameters to `UniversityMember`. That input parameter is then not created as a field within the `Person` class.

Concurnas automatically generates getters and setters for all fields prefixed with a tilde (`~`) character. By the way, a minus sign (`-`) generates only a getter, and a plus sign (`+`) generates only a setter. Here is an example:

```
p1 = new Person("dave", "brown", 1970)
name = p1.firstname //equivalent to: name = p
p1.firstname = "newName" //equivalent to: pe.
```

In Concurnas, as in many modern JVM languages, the equality operator (`==`) behaves as most developers expect it should, in assigning equality by *value* as opposed to by *reference*. The following code is valid:

```
p1 = Person("dave", "brown", 1970)
p2 = p1@ //deep copy of p1

assert p1 == p2        //different objects bu
assert not (p1 &== p2) //equality by referenc
```

Notice above that the code also makes use of the `@` copy operator. This operator provides a deep copy of the object referenced on its left side.

Concurnas automatically generates `hashcode` and `equals` methods for all classes. In this way, they can be used easily in common data structures such as sets and maps, for example:

```
p1 = new Person("dave", "brown", 1970)
myset = set()
myset.add(p1)

p2 = new Person("dave", "brown", 1970)
assert p2 in myset
```

As mentioned previously, Concurnas supports tuples. It also supports traits, which lets developers create classes via composition as opposed to creating classes exclusively via inheritance.

Concurnas includes many features traditionally seen in functional programming languages, such as method references and partial functions:

```
def plus(a int, b int) => a + b

//function reference
funcRef (int, int) int = plus&(int, int)

//called just as a normal function
result = funcRef(2, 3) //=> 5

//partial function
plusone (int) int = plus&(int, 1)

//also called just as a normal function
result = plusone(10) // ==> 11
```

In addition, pattern matching is supported. This is a very powerful mechanism that is similar to the `switch` statement in Java but much more powerful:

```
def matcher(an Object){
    match(an){
        Person(yob < 1970) => "Person. Born:
        Person => "A Person" //check if it is
        int; < 10 => "small number" //check i
        int => "another number"
        x => "unknown input" //match all othe
    }
}

res = matcher(x) for x in [Person("dave", "br
//res ==> [Person. Born: 1829, A Person, unkn
```

Pattern conditions are checked from first to last. They support matching against types with optional guards. Guards may also be specified on object fields.

Other aspects of functional programming that are provided in Concurnas include type inference, which was explored earlier, and lambdas and lazy evaluation.

Working with data

In addition to data science and numerical computing, most enterprise solutions involve working with data in one form or another. To this end, Concurnas provides strong support for working with common data structures. Let's define some now:

```
anArray = [1 2 3 4 5 6]
aList   = [1,2,3,4,5,6]
aMatrix = [1 2 3 ; 4 5 6]
aMap    = {"one" -> 1, "two" -> 2, "three" ->
```

Some supported operations include the following:

```
cont     = "one" in aMap //checking for a val
longNames = aMap[key] for key in aMap if key.

arrayValue = anArray[2]     //individual valu
subarray   = anArray[4 ...] //a subarray; [5
```

Concurnas provides support for list comprehensions, which are essentially syntactic sugar for `for` loops. They permit guard statements to be attached to them, such as `i mod 2 == 0` in this example:

```
ret = i+10 for i in aList if i mod 2 == 0
```

Concurnas also supports vectorization, which is a nice technique for when you need to perform the same element-wise operation on each component of a list or n-dimensional array. For example, if you wish to calculate `y = x**2 + 1` for each element `x` of a matrix, you can do so as follows:

```
mat = [1 2 ; 3 4]
mat2 = mat^*2 + 1 //==> [3 5 ; 7 9]

mat^^*2 + 1 //in-place vectorized operation
```

Numerical ranges have first-class citizen support and since their values are created on a lazy basis, infinite ranges are even possible:

```
numRange = 0 to 10           //a range of: [0
tepRange = 0 to 10 step 2    //a range of: [0
revRange = tepRange reversed //a reversed ran
decRange = 10 to 0 step 2    //a range of: [1
infRange = 0 to              //an infinite se
steInfRa = 0 to step 2       //a stepped infi
decInfRa = 0 to step (-1)    //a stepped infi

val = x for x in numRange    //list comprehen
check = 2 in numRange        //checking for t
```

Installing Concurnas

To start using the language, you must first install Java JDK 1.8 or later. Then you can install Concurnas.

The Concurnas download site provides an installer for Windows and a zip file for other platforms. By the way, on Linux, you should use the SDKMAN! SDK manager to install Concurnas via the following command:

```
$ sdk install concurnas
```

Once the JDK and Concurnas are installed, create a simple "hello world" application called `hello.conc`:

```
for(x in 1 to 5){
    System.out.println("hello world {x}")!
}
```

Concurnas follows the standard two-phase compilation and execution process seen in languages such as Java. Once you have saved your `hello.conc` code file, you can compile the code via the `concc` command, as follows:

```
$ concc hello.conc
```

And you can execute it via the `conc` command:

```
$ conc hello

hello world 1
hello world 2
hello world 5
hello world 4
hello world 3
```

Your output may differ from the output shown above because execution is performed concurrently!

As an alternative to using the two-phase compilation and execution approach, you can make use of the Concurnas Read-Evaluate-Print-Loop (REPL). It behaves just as Java JShell and can be spawned by passing no arguments to the `conc` command:

```
$ conc
Welcome to Concurnas 1.14.020 (Java HotSpot(T
Currently running in REPL mode. For help type

conc>
```

You're now all set to start leveraging the power of Concurnas for your next JVM project.

Conclusion

This article introduced the capabilities of the Concurnas programming language and showed how Concurnas can be used for building concurrent and parallel systems by making use of modern-day CPU and GPU hardware architectures. It explored various Concurnas features taken from the most

popular programming paradigms of today and showed how Concurnas is an ideal language for working with data and for leveraging existing code written in other JVM languages. To learn more about the language, the Concurnas website is an excellent place to start.

Jason Tatton

Jason Tatton (@concurnas) is the creator of the Concurnas programming language and the founder of Concurnas Ltd. Tatton has written and led teams developing algorithmic trading systems for some of the world's most prestigious investment banks including Bank of America, Merrill Lynch, Deutsche Bank, and J.P. Morgan. He is passionate about technology, programming, and making Concurnas the best programming language it can be.

Share this Page