ParaSail: Less is More with Multicore

S. Tucker Taft, ***Director of Language Research,*** **AdaCore**

Parallel multicore programming should be easier than it is. What makes it such a struggle? Most programming languages were designed without parallel programming in mind. Attempts to add on parallel programming features, or parallel libraries, are doomed to a struggle against the impediments built into the original languages.

What are the impediments to easy parallel programming? With apologies to David Letterman, we would identify the following as the biggest impediments to doing things efficiently and safely in a highly parallel multicore world:

- Global Variables
- Garbage-Collected Global Heap
- Parameter Aliasing
- Run-Time Exception Handling
- Explicit Threads
- Explicit Lock/Unlock
- Explicit Wait/Signal
- Race Conditions

and worst of all …

- Pointers

In what sense are these impediments, and how do we avoid them?

ParaSail (*Parallel Specification and Implementation Language*) [1] is a new parallel programming language with a familiar class-and-interface-based object-oriented programming model, but designed specifically to eliminate impediments to easy parallel programming. In ParaSail, it is easier to write parallel algorithms than sequential ones. If sequential execution is needed, the programmer has to work harder. The default execution semantics are parallel. Pointers are eliminated in favor of expandable and shrinkable objects. Race conditions are eliminated at compile-time, as are other possible run-time errors such as use of null values or uninitialized data, indexing outside of an array, numeric overflow, dangling references, etc. The net effect is a language that feels familiar, is a pleasure to program in, but nevertheless results in highly parallel, race-condition-free, well-structured programs.

Here is a simple recursive ParaSail function which counts the number of words in a string `S`, using `Separators` as the set of characters that separate one word from another:

```
func Word_Count
  (S : Univ_String;
   Separators : Countable_Set<Univ_Character> := [' '])
  -> Univ_Integer is
    // Return count of words separated by given set of separators
    case Length(S) of
      [0] => return 0; // Empty string
      [1] =>
        if S[1] in Separators then
            return 0; // A single separator
        else
            return 1; // A single non-separator
        end if;
      [..] =>         // Multi-character string; divide and conquer
        const Half_Len := Length(S)/2;
```

```
            const Sum := Word_Count(S[1 .. Half_Len], Separators) +
              Word_Count(S[Half_Len <.. Length(S)], Separators);

            if S[Half_Len] in Separators
              or else S[Half_Len+1] in Separators then
                return Sum;    // At least one separator at border
            else
                return Sum-1;  // Combine words at border
            end if;
        end case;
    end func Word_Count;
```

One interesting thing about this ParaSail function is that it doesn't appear to be explicitly parallel, but in fact the basic semantics of ParaSail imply that the two recursive calls of `Word_Count` used to initialize the value of `Sum` can be evaluated in parallel. Those recursive calls then continue the *divide-and-conquer* approach to counting the words in the string, with the net effect that for a string of length N, N *picothreads* are spawned, with their results added together in a computation tree of depth $\log_2(N)$, providing a potential speedup of order $N/\log_2(N)$ relative to a sequential algorithm.

In the remainder of this article we will go through each of the impediments mentioned above, discuss the problems they create, and illustrate how parallel programming in ParaSail is simplified by doing without them.

No Global Variables

Many of us have been warned of the evils of global variables in our early programming courses, but nevertheless, global variables are widely used in most programs. Sometimes they aren't globally visible, but nevertheless, anything that represents variable global state, such as a *static* variable in *C*, *C++,* or *Java*, or a singleton object in *Scala* [2], can create the same kinds of problems associated with unexpected side-effects, hidden algorithmic coupling, etc. In a sequential program, global variables are often considered bad practice. In a parallel program, global variables can create nasty race conditions, or synchronization bottlenecks.

So how do you live without global variables? The ParaSail model is very straightforward – a function may only update data that are part of a **var** parameter passed to the function. This means that the specification of a function indicates all of the possible effects of the function. There are no mysterious side effects to worry about. A typical function that is often implemented using a global variable is a (pseudo) random number generating function. In ParaSail, a module that provides random numbers could have the following interface:

```
interface Random<> is
    func Start(Seed : Univ_Integer := 1) -> Random;
      // Start a new random number sequence

    func Next(var Seq : Random) -> Univ_Integer;
      // Get next value in random number sequence
end interface Random;
```

This could be used as follows:

```
func Roll_The_Dice(var Ran : Random) -> Univ_Integer is
  // return the sum of rolling two dice
    const First := Next(Ran) mod 6 + 1;
    const Second := Next(Ran) mod 6 + 1;
    return First + Second;
end func Roll_The_Dice;
```

If desired, the first parameter in a call in ParaSail may generally be moved to the front, so instead of `Next(Ran)`, one could write `Ran.Next()`. Note in the above example the types for `First` and `Second` are inferred from context, though they could be given explicitly. Below we show explicit types, and the alternative call notation:

```
      const First : Univ_Integer := Ran.Next() mod 6 + 1;
      const Second : Univ_Integer := Ran.Next() mod 6 + 1;
```

In the above declarations we are using `Univ_Integer`, a universal integer type that supports arbitrarily large integers. In many cases, the programmer will instead choose to declare a new integer type with a specified range, such as:

```
      type Dice_Sum is Integer<2..12>;
      func Roll_The_Dice(var Ran : Random) -> Dice_Sum is ...
```

Doing this can provide more efficiency and additional consistency checking, as well as more information to the user of the routine.

No Garbage-Collected Global Heap

Using a single garbage-collected heap is a common feature of languages like *Java* and other JVM languages, *C#* and other .NET languages, *Eiffel*, *ML*, and many other modern languages. But a single global heap can be a problem for a highly parallel application. Due to the importance of the cache to performance, locality of reference is important within a single thread, while separation is important for threads operating in parallel to avoid cache interference effects, such as *false sharing [3]*. Unfortunately, a single global heap can produce the worst of both worlds – the globally-shared heap results in the objects manipulated by a single thread being potentially spread all over memory, while objects created by concurrently executing threads being allocated in neighboring memory locations.

So what is the alternative to a global, garbage-collected heap, without forcing users to resort to completely manual storage management? ParaSail uses a variant of *region-based storage management*, pioneered in the *Cyclone* language [4], where objects are allocated out of *regions* (which are essentially local heaps), and the storage for a region is reclaimed automatically when leaving the scope with which the region is associated. In ParaSail, every object lives within the region associated with the stack frame of the function in which it is declared. ParaSail objects can grow and shrink in size over their lifetime, and the storage needed for the growth comes out of the region, and the storage released by shrinkage goes back into the region. No data is shared between objects, so the allocation and release happen automatically and immediately. There is no need for an asynchronous garbage collection process; there is never any *garbage* to collect.

When a function creates and returns an object to its caller, it automatically allocates it from the region determined by the caller. For example, the `Start` function of the above `Random` module creates and returns an object that can produce a sequence of random numbers. The implementation of a ParaSail module like `Random` is provided using the **class** construct, which defines any private data and operations, as well as the implementation of the exported operations like `Start` and `Next`:

```
      class Random is
        // class defining Random module's data and operations
          const Mult := ...
          const Modulus := ...
          var Last_Value : Univ_Integer;
        exports
          func Start(Seed : Univ_Integer := 1) -> Random is
            // Start a new random number sequence
              return (Last_Value => Seed mod Modulus);
          end func Start;
          func Next(var Seq : Random) -> Univ_Integer is
            // Get next value in random number sequence
              Seq.Last_Value := Seq.Last_Value * Mult mod Modulus;
              return Seq.Last_Value;
          end func Next;
      end class Random;
```

The **return** statement in the implementation of `Start` creates the object to be returned by specifying the values of its components (in this case, only `Last_Value` – `Mult` and `Modulus` are global constants in this implementation of `Random`). The storage for the object will be allocated out of the region determined by the caller. This might likely be the region associated with the *main* function of the program, such as:

```
    func Play_A_Game(var IO) is
      // Roll the dice 100 times and see what happens
        var Ran := Random::Start();

        for I in 1..100 loop
            const Roll := Some_Game::Roll_The_Dice(Ran);

            IO.Println("Rolling the dice gave " | Roll);
            case Roll of
              [2]  => IO.Println("Snake eyes");
              [12] => IO.Println("Box cars");
              [7]  => IO.Println("Lucky seven");
              [..] => IO.Println("Boring");
            end case;
        end loop;
    end func Play_A_Game;
```

Note that this main function receives an `IO` parameter. This provides access to standard input and output files, as well as other parts of the underlying file system. We are presuming that there is a module `Some_Game` in which our function `Roll_The_Dice` appears.

No Parameter Aliasing

If a function takes two or more parameters that are passed by reference, and at least one of them can be updated within the function, there is the possibility that changing the updatable parameter might affect the value of the other one. Having two different names that refer to the same object is called *aliasing*, and if it is possible between parameters, it can interfere with optimizations inside the function. In a pervasively parallel language, this kind of potential parameter aliasing can interfere with parallelizing the code of the function.

In ParaSail, the compiler disallows parameter aliasing at the point of call, which ensures that every function can be fully parallelized. ParaSail actually goes further to ensure that it can parallelize the computation of all expressions. This is best illustrated by example. The following alternative implementation of `Roll_The_Dice` is illegal in ParaSail:

```
    func Roll_The_Dice(var Ran : Random) -> Univ_Integer is
        // the following is illegal in ParaSail:
        return (Next(Ran) mod 6 + 1) + (Next(Ran) mod 6 + 1);
    end func Roll_The_Dice;
```

The above is illegal because the two operands of the outermost "+" both involve the `Ran` variable, which if evaluated in parallel, might result in unsafe concurrent updates to the `Ran` variable. Compare this with the `Word_Count` example given above:

```
    const Sum := Word_Count(S[1 .. Half_Len], Separators) +
       Word_Count(S[Half_Len <.. Length(S)], Separators);
```

Here we have two recursive calls on `Word_Count` on either side of a "+" operator. This is fine in this case for two reasons. First of all, `Word_Count` is not updating its parameters, so there are no issues with aliasing. But even if the `S` parameter of `Word_Count` were a **var** parameter, the expression would have been legal, because non-overlapping *slices* of the `S` array, `S[1..Half_Len]` and `S[Half_Len<..Length(S)]`, are being passed to the two recursive invocations of `Word_Count`. (The notation `X <.. Y` in ParaSail represents the half-open interval `(X,Y]`, which includes all values `V` such that `X < V <= Y`; this has no overlap with the interval `1..X`, which includes all the values `V` such that `1 <= V <= X`.)

The net effect of the rules in ParaSail disallowing parameter aliasing, and more strictly, disallowing two references to the same variable in different parts of the same expression, if either one might result in an update, is that any (legal) ParaSail expression can be evaluated in parallel. The compiler uses heuristics to decide whether it is more efficient to actually evaluate the operands of an expression in parallel, but the semantics are such that parallel evaluation would always be safe.

No Run-Time Exception Handling

Many modern languages provide support for the *raising* or *throwing* of exceptions in an inner scope, and then *handling* or *catching* them in an outer scope. Run-time exception handling serves several purposes. First, it means that errors cannot simply be ignored, unless the programmer takes explicit action to do so by handling and then ignoring the exception. Second, exceptions help avoid a race condition between checking on the state and then acting on the state, in the case where the state might be updated in the interim. For example, checking whether a file exists and then opening it as two separate operations leaves room for the file to be deleted between the test and the actual open. The problem is eliminated if the open operation itself performs the check and either raises an exception indicating the file doesn't exist, or opens the file and thereby prevents it from being deleted until it is later closed. Finally, exceptions are also useful for signaling bugs or unanticipated situations, such as indexing outside the bounds of an array, attempting to read an invalid data value, dereferencing a null pointer, or running out of memory. The handling for such exceptions is generally limited to some kind of reset and restart, perhaps in a degraded mode.

Run-time exception handling, while clearly useful, has its downsides. An exception handler creates the potential for a large number of additional paths of execution through a function, and these paths are generally very hard to test thoroughly. It may be possible for an exception handler to refer to an object that has not been properly initialized, if the exception was raised before or during the initialization of the object. And in a highly parallel environment, exception handling is notoriously troublesome, because it is not always clear what should happen when one thread raises an exception but does not handle it locally, while other concurrent threads are proceeding normally. Should the whole computation be abandoned? Should the exception be propagated when all of the other concurrent threads complete their work? Should the exception be swallowed and ignored?

So how does ParaSail provide solutions for the situations where run-time exception handling is useful, without incurring some of the downsides of run-time exception handling? The primary solution in ParaSail is that potential run-time errors are identified at compile time. For example, the ParaSail compiler will complain if an array is indexed with a value that is not known to be within the array bounds. If this value was passed in as a parameter to a function, then the implementor of the function is obliged to specify an appropriate *type* or *precondition* for the parameter that ensures it is within the appropriate range. At the call point, the actual parameter must be provably within the appropriate bounds, or again the compiler will complain. The requirement can be propagated further out through further use of appropriate types or preconditions, or an explicit check can be performed on the range, with the call only being performed if the range is satisfied, with some alternative action taken otherwise. Effectively the programmer is doing "exception handling" at *compile time*, by propagating requirements from a function to its callers, and handling them with explicit checks if necessary at the appropriate level. For example, here is the interface to a simple Stack module:

```
interface Stack<Component is Assignable<>> is
    func Create(Max : Univ_Integer) -> Stack;
    func Max_Stack_Size(S : Stack) -> Univ_Integer;
    func Count(S : Stack) -> Univ_Integer;
    func Push(var S : Stack; C : Component);
    func Pop(var S : Stack) -> Component;
end interface Stack;
```

and here is a **class** that might define this `Stack` module:

```
class Stack is
    const Max_Len : Univ_Integer;
    var Cur_Len : Univ_Integer;
    var Data : Basic_Array<optional Component>;
  exports
    func Max_Stack_Size(S : Stack) -> Univ_Integer is
        return S.Max_Len;
    end func Max_Stack_Size;

    func Count(S : Stack) -> Univ_Integer is
        return S.Cur_Len;
    end func Count;

    func Create(Max : Univ_Integer) -> Stack is
        return (Max_Len => Max, Cur_Len => 0,
          Data => Create(Max, null));
    end func Create;

    func Push(var S : Stack; X : Component) is
        S.Cur_Len += 1;
        S.Data[S.Cur_Len] := X;
    end func Push;

    func Pop(var S : Stack) -> Result : Component is
        Result <== S.Data[S.Cur_Len];
        S.Cur_Len -= 1;
    end func Pop;
end class Stack;
```

The ParaSail compiler will complain about the line in the `Push` function that assigns `X` into `S.Data[S.Cur_Len]`, since there is nothing that prevents `S.Cur_Len` from being outside the bounds of the `S.Data` array (which are `1..S.Max_Len`). A similar complaint will be made about the line in `Pop` that retrieves the value of `S.Data[S.Cur_Len]`. To handle this complaint, the programmer can add *preconditions* to `Push` and `Pop`. In ParaSail, preconditions, postconditions, assertions, etc., are all specified using a notation inspired by *Hoare logic*, using one or more Boolean expressions enclosed in {...}. So adding a precondition to the specification for `Push` would produce:

```
func Push(var S : Stack; C : Component)
  {Count(S) in 0 ..< Max_Stack_Size(S)};
```

With this change, the body of the `Push` **func** will now compile without complaint. However, if we write some code to use `Push`:

```
var Stk : Stack<Integer<1..100>> ::= Create(10);
```

```
Stk.Push(42);
```

the ParaSail compiler will now complain about the call on `Push`, because it has no way of determining whether the precondition on `Push` is satisfied (without peaking inside the **class** defining the `Stack` module, and that isn't allowed – callers may only depend on information provided in the **interface**). So now we have to provide some *postconditions* for the operations of the `Stack` module. In particular we need to say what the `Count` and `Max_Stack_Size` are of a newly created stack object, and how they might change as a result of a `Push` or a `Pop`. Here is a fully annotated **interface** for the `Stack` module:

```
interface Stack<Component is Assignable<>> is
    func Max_Stack_Size(S : Stack) -> Univ_Integer;
    func Count(S : Stack) -> Univ_Integer
      {Count in 0..Max_Stack_Size(S)};
    func Create(Max : Univ_Integer {Max > 0}) -> Stack
      {Max_Stack_Size(Create) == Max; Count(Create) == 0};
    func Push
      (var S : Stack {Count(S) < Max_Stack_Size(S)};
       X : Component) {Count(S') == Count(S) + 1};
    func Pop(var S : Stack {Count(S) > 0}) -> Component
      {Count(S') == Count(S) - 1};
end interface Stack;
```

Note that we have been able to simplify the precondition on `Push` because the postcondition on `Count` ensures that it is always in the range `0..Max_Stack_Size(S)`. Looking at the above annotations we see that the `Create` function returns a stack of the given `Max` size, but with no items on it initially (`Count` is zero). `Push` adds an item to the stack as reflected by the change in Count (in a postcondition, `S'` refers to the state of the **var** parameter `S` *after* the operation is complete, while `S` by itself refers to the initial state). `Pop` removes an item. Syntactically, postconditions must come after any preconditions, and after the **->** if any. Postconditions may refer to the *post*-state of the **var** parameters, as well as to the outputs, if any (the outputs are specified after the **->** symbol, and if there is only one output, it may be referred to by the name of the **func** itself – see postconditions for `Create` and `Count` above as examples).

The above pre- and postconditions should allow the user of the `Stack` module to use its operations properly, and give the ParaSail compiler enough information to check that calls on `Push` and `Pop` will not cause a stack overflow or underflow. The ParaSail compiler will also check when compiling the **class** that defines `Stack` that the operations satisfy the postconditions, given the preconditions. The programmer may need to provide additional annotations, such as an *invariant* on the state of the components of the object, to enable the compiler to prove that the postconditions are always satisfied.

In addition to those features discussed above, ParaSail has other features that help obviate the need for run-time exception handling. In particular, every type has a **null** value in addition to the "regular" values. The **null** value can be used for situations like opening a file – a **null** file handle may be returned if the file does not exist. The **null** value can be checked for by using `X is null` or `X not null`. However, a **null** value is only permitted when the qualifier **optional** is given when specifying the type of an object, component, parameter, or result. In the absence of the qualifier **optional**, there is an implicit assertion that the value of the object, component, parameter, or result satisfies `{X not null}`. In the above **class** defining the `Stack` module, we used **optional** when specifying the component type of the Data array to indicate that the values in the array may have the **null** value. However, we did *not* specify **optional** for the `Component` parameter of `Push`, or the `Component` result of `Pop`. This means that the ParaSail compiler will need to prove that when `S.Data[S.Cur_Len]` is retrieved in `Pop` it is not **null**. It turns out this will require that the programmer write an *invariant* on the `Data` component of the following form:

```
{(for all I in 1..Cur_Len => Data[I] not null)}
```

If such an invariant is specified in the **class,** it will be implicitly added to every precondition, and will need to be proved in addition to any postcondition, for every operation of the `Stack` module. With this invariant, the ParaSail compiler can prove that the value returned by `Pop` will not be **null**.

One final important use for run-time exceptions is to indicate when some resource is exhausted, such as running out of disk or stack space. In ParaSail a computation can be split into a part that does the real work, and a part that monitors for resource exhaustion or other important event. When the event occurs, the overall computation can be terminated with a **return** or **exit** statement which specifies a value for the computation that indicates the cause for the early termination.

No Explicit Threads, Lock/Unlock, Wait/Signal

Languages such as *Java* and *Ada* that have some built-in support for concurrent programming traditionally have provided the explicit notion of a *thread* or a *task*, along with some mechanism for synchronizing the threads, using locks and signals of some sort. In *Java*, locking is performed using *synchronized* methods or blocks. Unlocking is automatic on leaving the method or block. In *Ada*, locks are associated with *protected objects*, and locking and unlocking are automatic as part of calling any operation defined as part of defining the protected type.

In *Java*, waiting and signaling are explicit, but the conditions associated with the waiting are not specified explicitly, but rather are implicit in the logic of the threads using the wait and signal operations. In *Ada*, waiting and signaling are implicit in special operations on protected objects called *entries*. These operations explicitly specify an *entry barrier* that must be true before a caller of the entry is permitted to execute the entry body. A caller will be suspended on an associated *entry queue* automatically until the *barrier* expression is true and no other task is currently operating on the object, at which point the entry body will be executed with a guarantee that the *barrier* expression is true at the start of the execution of the body. By making the locking, unlocking, waiting, and signaling implicit in the language construct, various synchronization errors are automatically eliminated.

ParaSail takes this approach one step further. As we have seen in the `Word_Count` example, threads or tasks are themselves implicit in the language, and whether an "actual" thread of some sort is created is a decision left to the compiler, based on heuristics about the overhead of creating a thread relative to the potential savings from parallel execution. In other respects, ParaSail adopts a model similar to that of *Ada*, where an object as a whole is either designed for concurrent access or not; only if a ParaSail object is a **concurrent** object will simultaneous access by parallel computations be permitted.

A ParaSail object becomes **concurrent** either by being an instance of a type that is defined by a **concurrent** module, or by having the **concurrent** qualifier specified explicitly at the point the object is declared. When an individual object is declared as **concurrent**, effectively what happens is a **concurrent** module is implicitly created to act as a *wrapper* of the specified non-concurrent module defining the object.

Here is the **interface** of a simple (explicitly) **concurrent** module:

```
concurrent interface Locked_Box<Content_Type is Assignable<>> is
    func Create(C : optional Content_Type) -> Locked_Box;
        // Create a box with the given content
    func Put(queued var B : Locked_Box; C : Content_Type);
        // Wait for the box to be empty (i.e. null)
        // and then Put something into it.
    func Content(locked B : Locked_Box) -> optional Content_Type;
        // Get a copy of current content
    func Get(queued var B : Locked_Box) -> Content_Type;
        // Wait until content is non-null,
        // then return it, leaving it null.
    end interface Locked_Box;
```

This **concurrent** module provides a simple box into which an object of the specified Content_Type may be `Put` and subsequently retrieved, either non-destructively with `Content` or destructively with `Get`. Any object of a type defined by this module is a **concurrent** object, and arbitrary simultaneous access is permitted from parallel computations. Here is a simple example using this box, where we have 100 concurrent picothreads, half trying to Put something in, and the other half trying to get something out:

```
    func Cross_Talk(var IO) is
        var B : Locked_Box<Integer<1..50>> := Create(null);
        for I in 1..100 concurrent loop
            if I <= 50 then
                B.Put(I);
            else
                IO.Println("Thread " | I | " got " | B.Get());
            end if;
        end loop;
    end func Cross_Talk;
```

Here is a possible implementation of the `Locked_Box` module:

```
    concurrent class Locked_Box is
        var Content : optional Content_Type;
      exports
        func Create(C : optional Content_Type) -> Locked_Box is
            // Create a box with the given content
            return (Content => C);
        end func Create;

        func Put(queued var B : Locked_Box; C : Content_Type) is
          queued until B.Content is null then
            // Wait for the box to be empty (i.e. null)
            // and then Put something into it.
            B.Content := C;
        end func Put;

        func Content(locked B : Locked_Box)
          -> optional Content_Type is
            // Get a copy of current content
            return B.Content;
        end func Content;

        func Get(queued var B : Locked_Box)
          -> Result : Content_Type is
          queued while B.Content is null then
            // Wait until content is non-null,
            // then return it, leaving it null.
            Result <== B.Content;
        end func Get;

    end class Locked_Box;
```

Note the *dequeue conditions* specified for the operations `Put` and `Get` which take a **queued** operand. This condition specifies what needs to be true (if **queued until**) or false (if **queued while**) before the code of the operation is executed. As with an *Ada* entry barrier, ParaSail guarantees that the dequeue condition is satisfied at the point where the body of the **func** begins execution. In some sense a dequeue condition is a kind of *synchronizing* precondition; it is a precondition that need not be satisfied when first called, but the operation will not actually execute until it is satisfied.

Also note that we use the `<==` operation in the `Get` **func**; this ParaSail operation *moves* the right-hand side into the left-hand side, leaving the right-hand side **null.** This operation minimizes copying when the left- and right-hand sides are associated with the same region. ParaSail also provides a `<=>` operation, which *swaps* the left- and right-hand sides, again minimizing copying when possible. The normal `:=` assignment operation *copies* the right-hand side into the left-hand side (unless the right-hand side is an aggregate or the result of a function call, in which case the result is *moved* in).

In addition to the **locked** and **queued** operations exemplified above, a **concurrent** object may also provide *lock-free* operations. Such operations are limited to using *lock-free* primitives such as atomic load or store, atomic test-and-set, or atomic compare-and-exchange. The ParaSail standard library provides a set of **concurrent** modules that define lock-free primitives, and more can be implemented using low-level ParaSail features.

No Race Conditions

A *race condition* is a situation in which two concurrent computations manipulate the same object without adequate synchronization, resulting in potentially unexpected or undefined behavior. ParaSail eliminates the possibility for race conditions, by compile-time checks as described above under the ***No Parameter Aliasing*** section, plus the lack of global variables, and the rules for **concurrent** objects. Essentially ParaSail eliminates race conditions *by construction*.

A more general definition of *race condition* might be any computation whose result depends on the relative timing of two sub-computations. ParaSail *does* allow for such computations. However, these kinds of *intentional* race conditions will only occur if the programmer creates one or more **concurrent** objects, and then manipulates them concurrently from parallel computations. If no **concurrent** objects are manipulated in parallel, then the result of the ParaSail program is *deterministic*. For example, the Word_Count function we showed at the beginning of this article is highly parallel, but its result is independent of the relative rate at which the various parallel sub-computations are performed.

Of course **concurrent** objects may be essential in a given computation, and **concurrent** objects are often used to represent the *external* environment, such as a file system, database, or external process. If interactions with the external environment are performed from parallel sub-computations, then clearly the results will depend on the relative rate of these sub-computations. This is the very nature of a real-time or interactive application, and so for these any sort of *determinism* is already compromised by the non-deterministic nature of a real-time or interactive environment. In ParaSail, the programmer is in control of the amount of non-determinism in the application, and the results will never be undefined due to inadequate synchronization; concurrent access is only permitted to **concurrent** objects, which have well-defined semantics in the presence of concurrent callers.

No Pointers

We end with a discussion of pointers, perhaps the worst impediment to easy parallel programming. Why are pointers so bad? Because they interfere with the critical *divide-and-conquer* approach to parallel programming. To divide and conquer a problem, it is necessary to be able to separate one part of the problem from another. If pointers are used to represent the fundamental data structures, cleanly dividing one half of a data structure from another is not so easy. The programmer may “know” that the structure is a binary tree, but what happens if through some bug or undocumented “feature” there is actually some sharing happening between parts of the tree? Is it safe to send one thread off to update the left “half” of the tree while simultaneously sending another to update the right “half,” if there is some possibility of them meeting up somewhere later in the middle of their updates?

So how does ParaSail avoid pointers? If we look back on the sections above, we see that ParaSail is a flexible language with a familiar class-and-interface-based object-oriented programming model, but it has no need for pointers. The region-based storage management model, the availability of a **null** value for any type, the ability to assign a non-null value into a previously **null** component (or vice-versa), makes it possible for objects to *grow* and *shrink* over their lifetime without any explicit use of pointers. Yes perhaps behind the scenes the ParaSail implementation is using pointers, but by eliminating them from the language-level semantics of the language, we avoid essentially all of the problems that pointers can create for a parallel program.

Other ParaSail features we have not discussed above, such as the uniform ability to define how the indexing and slicing operations (`A[I]` and `A[I..J]`) work for any *container*-like type, mean that constructs like *directed graphs* are easily represented in ParaSail without pointers; the graph would be a container object, a node identifier would be used for indexing into the graph container, and edges in the graph would be represented by pairs of node identifiers. Here is an example of a directed graph **interface**:

```
interface DGraph<Element is Assignable<>> is
    type Node_Id is new Integer<1..10**6>;
    func Create() -> DGraph;
    func Add_Node(var DGraph; Element) -> Node_Id;
    op "indexing"(DGraph; Node_Id) -> Element;
    func Add_Edge(var DGraph; From, To : Node_Id);
    func Successors(DGraph; Node_Id) -> Set<Node_Id>;
    func Predecessors(DGraph; Node_Id) -> Set<Node_Id>;
end interface DGraph;
```

If an "indexing" operator is defined as above, then we can use the `A[I]` notation with objects of a type defined by the given module. There are other operators that if defined, give access to other ParaSail syntax. In general ParaSail uses a *syntactic sugaring* approach which turns uses of special syntax such as indexing, slicing, iterators, etc., into a series of calls on specific operators. If the operators are defined for the relevant types, the syntax is allowed.

Note that we are not giving parameter names for most of the above operations. Parameter names are generally optional on the specification, and if the type names of the inputs are unique, the type names may be used to refer to the corresponding input, both at the call site and within the body of the operation.

We will leave implementing the DGraph module as an exercise for the reader. We will also post an example implementation on the ParaSail blog [1].

ParaSail Summary

The goal of ParaSail is to make parallel programming easy and safe. The main innovations in ParaSail are represented by what impediments have been *eliminated* in the name of that goal, and in the flexibility and ease of use of what is left. Programming in ParaSail is a pleasure, both because so many of the nasty debugging problems are eliminated from the start by the fundamental semantics of the language, and because the ability to use parallelism in a simple and efficient way can make the expression of the solution to a problem more natural. ParaSail's region-based, pointer-free data structuring model makes it easy to create data structures that naturally support a divide-and-conquer approach, and the checking provided at compile time eliminates many of the run-time failures and race conditions that can dramatically slow down the development and testing of a complex, parallel application.

Further Reading

[1] ParaSail blog: http://parasail-programming-language.blogspot.com

[2] Scala language: http://www.scala-lang.org

[3] False sharing: http://iacoma.cs.uiuc.edu/iacoma-papers/false_sharing.pdf

[4] Cyclone region-based memory management:
http://www.eecs.harvard.edu/~greg/cyclone/papers/cyclone-regions.pdf