ParaSail

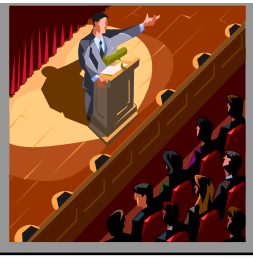
Parallel Specification and

Implementation Language

A Simplified Approach to Parallel Programming

S. Tucker Taft

June 2012



Outline of Presentation

- Why Design A New Language for High-Integrity Systems?
- Building on existing languages
- A Simplified Approach to Parallel Programming
- Building a program in ParaSail
- How does ParaSail compare?
- Open issues in ParaSail

Why Design A New Language for High-Integrity Systems?

- 80+% of safety-critical systems are developed in C and C++, two of the least safe languages invented in the last 40 years
- Every 40 years you should start from scratch
- Computers have stopped getting faster ----->
- By 2020, most chips will have 50-100 cores
- Advanced Static Analysis has come of age -- time to get the benefit at compile-time
- *It's what I do*

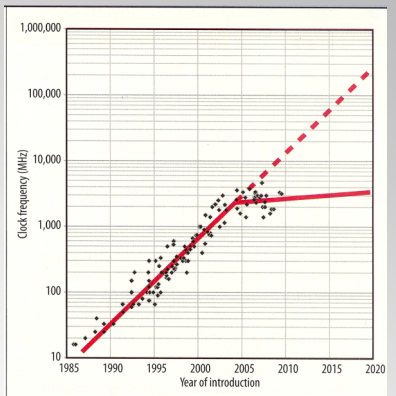


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Why a new language?
Computers have stopped getting faster

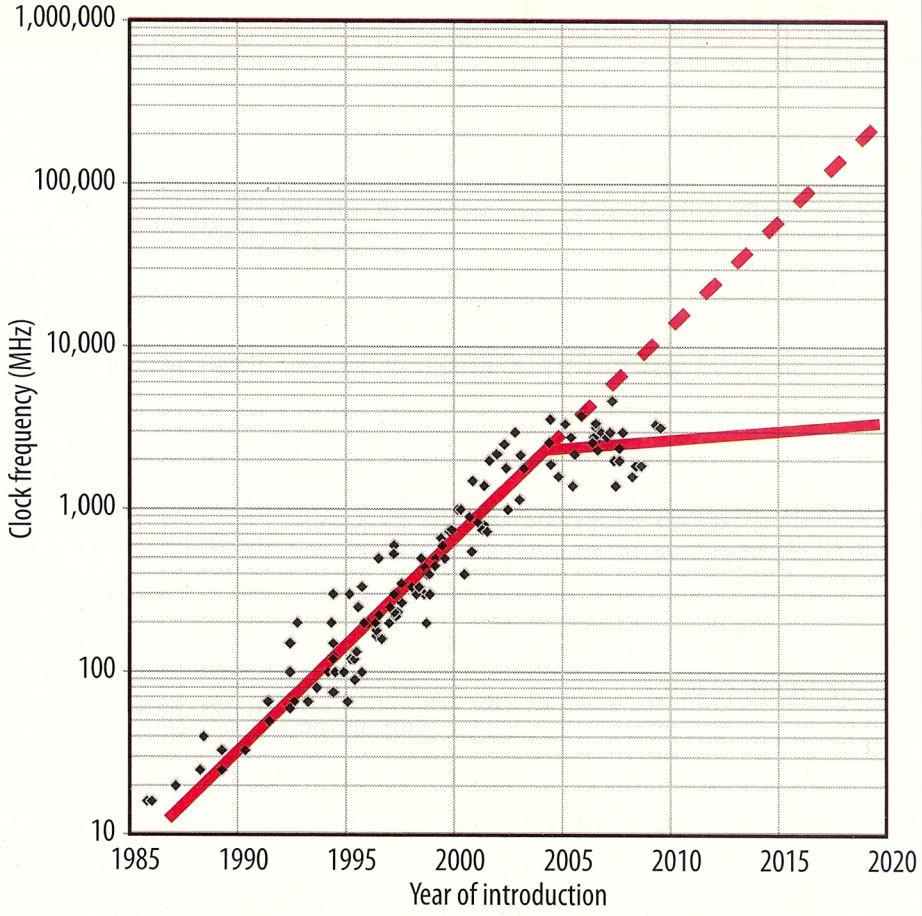


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy IEEE Computer, January 2011, page 33.

Building on Existing Languages

- Lisp, ML, and friends
  - Lisp, CLOS, Scheme, ML, OCaml, F#, Scala, Clojure
- More conventional languages
  - Pascal, Ada, Modula, Oberon, Cyclone, C++, Java, C#, Python
- Parallel programming languages
  - C*, HPF, CUDA, Cilk, OpenCL, CnC
  - Challenges both in coding and then debugging
- Languages with assertions, pre/postconditions, etc.
  - Eiffel, SPARK, Ada 2012, JML, SAL
  - Both for specification and for debugging
  - Manage the growing complexity and parallelism

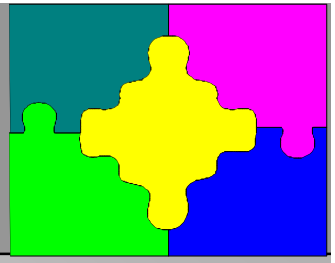
A Simplified Approach to Safe Parallel Programming

- Simplify/Unify
  - Smaller number of concepts, uniformly applied, all features available to user-defined types
  - Simplify to make *conceptual room* for parallelism and formalism
- Parallelize
  - Parallel by default
  - Have to work harder to force sequential execution
- Formalize
  - Assertions, Invariants, Preconditions, Postconditions integrated into the syntax
  - All checking at compile-time -- if compiler can't prove the assertion, program is illegal -- no run-time checking, no run-time exceptions

Simplify: Remove Impediments to Parallel Programming -- Any Tears?

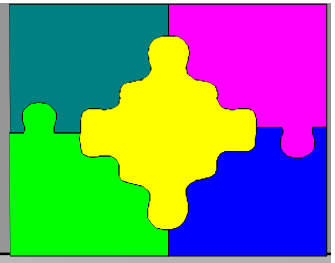
- No global variables
- No global, garbage-collected heap
- No pointers
- No hidden aliasing
- No special syntax reserved for built-in types
- No run-time exception handling
- No explicit threads, lock/unlock, wait/signal
- No race conditions

A Simplified Approach to Modules and Types



- Package, Template, Namespace, Class, Interface, Object, Record, Struct, Structure, Sig, Signature, Monitor, Task
  - Do we really need more than one of these?
  - What unifies these?
    - Their components are heterogeneous at compile-time
    - A symbolic selector identifies a component and determines its type.
  - Some are parameterized, some are not.
  - Some are modules, some are types, some are both.
  - How do operations fit in?
    - i.e. is it Object.Operation(...) vs. Operation(Object, ...)?

ParaSail approach for Modules/Types

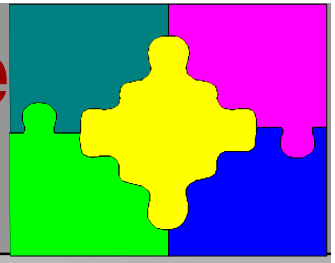


- All modules are parameterized (using <...> syntax)
  - Can share compiled code even though effectively *generic*
- All modules have an **interface** and (unless **abstract**) a **class** defining it; hierarchically named using "::"
  - May inherit operation *interfaces* from multiple modules
  - May inherit data and operation *code* from only one module
- Every **type** is an instance of a module, e.g.:
  - **type** Count **is new** Std::Integer<1..10>
  - **type** Color **is** Com::SofCheck::Enum< [#red, #green, #blue] >
  - Array<Employee, Indexed_By => Employee_ID>
- Types use structural equivalence unless marked **new**
  - I.e., in absence of **new**, identical parameterizations of same module produce same type (whether named or anonymous)

Example: N-Queens Interface

```
interface N_Queens <N : Univ_Integer := 8> is
   // Place N queens on an NxN checkerboard so that none of them can
   // "take" each other.  Return vector of solutions, each solution being
   // an array of columns indexed by row indicating placement of queens.

  type Chess_Unit is new Integer<-N*2 .. N*2>;
  type Row is Chess_Unit {Row in 1..N};
  type Column is Chess_Unit {Column in 1..N};
  type Solution is Array<optional Column, Indexed_By => Row>;

  func Place_Queens() -> Vector<Solution>
   {for all S of Place_Queens => for all C of S => C not null};
end interface N_Queens;
```

Other ParaSail Module/Type Features



- Given: **var** `Obj : T;`
  - `Obj.Op(...)` is equivalent to `T::Op(Obj, ...)`
  - No need for "`T::`" in general; compiler looks in all associated modules of operands for operation of given name
  - Operators like "`+`" treated uniformly, `Obj + X` is equivalent to "`+`"`(Obj, X)` and `T::`"`+`"`(Obj, X)` and `Obj.`"`+`"`(X)`
- Integer, Real, String, Character, Enumeration literals can be used with user-defined types
  - based on presence of "from_univ" operation(s) for type
  - all literals of a "universal" type
  - Univ_Integer (42), Univ_Real (3.141592653589793)
  - Univ_String ("Hitchhiker's Guide"), Univ_Character ('π')
  - Univ_Enumeration (#green)

A Simplified & Unified Approach to Arrays/Containers

- Collections/Containers: Array, Map/Hashtable, Tree, Set, Vector, Linked list, Sequence, ...
  - Elements are “key => value” or “key => is_present”
  - Homogeneous (at compile-time)
    - might be polymorphic at run-time (via a tag of some sort)
  - Iterators, indexing, slicing, combining/merging/concatenating
  - Empty container representation (e.g. “[]”)
  - Explicit “literal” instance, e.g.:
    - [2|3|5|7 => #prime, .. => #composite]
  - May grow or shrink over time
  - Region-based automatic storage management

ParaSail Approach for Containers

- `Container[Index]` for indexing
- `Container[A..B]` for slicing
- `[]` for empty container
- `[key1..key2=>val1,key3=>val3]` or `[val1,val1,val3]` for container aggregate
- `X|Y` for combining/concatenating/merging
- `C|=Y` or `C|=[key=>Y]` for adding Y to container C
- *User* defines operators "indexing", "[]", and "I=" and then compiler will create temps to support "X I Y" and "[...]" aggregates.

Expandable Containers Instead of Pointers

- Objects can be declared **optional** and can grow and shrink
  - Particularly useful for tree structures
  - Eliminates many of the common uses for pointers
- Generalized indexing into containers replaces pointers for cyclic structures
  - Similar to region-based storage management
  - `Region[Index]` equiv to `*Index`
    - presuming `Index` points into `Region`
- Short-lived references to existing objects are permitted
  - Returned by user-defined indexing functions, for example

Why and How to Parallelize?

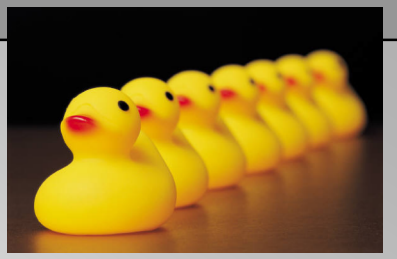
- Computers have stopped getting faster -- they are getting "fatter" -- more cores, not more GHz
- Programmers are lazy -- will take path of least resistance

⇒ The default should be parallel -- must work harder to force sequential evaluation.

⇒ Programmer mindset: there are 1000s of threads, and their goal is to use as many as possible.

- Compiler should prevent race conditions, and ideally, deadlock as well.

Simplified Approach to Parallelism in ParaSail



- Parallel by default
  - parameters are evaluated in parallel
  - have to work harder to make code run sequentially
- Easy to create even more parallelism
  - Process(X) || Process(Y) || Process(Z);
  - for I in 1..10 **concurrent** loop ... end loop;
- Lock-based and lock-free concurrent objects
  - Lock-based objects also support conditionally queued access
  - User-defined delay and timed call based on queued access
- No global variables
  - Operation can only access or update variable state via its parameters
- Compiler prevents aliasing and unsafe access to non-concurrent variables
  - Overall *pure value semantics* for non-concurrent objects

Expression and Statement Parallelism

- Within an expression, parameters to an operation are evaluated in parallel
  - `F(X) + G(Y)*H(Z)` -- F(X),G(Y),H(Z) evaluated concurrently
- Programmer can force parallelism across statements
  - `P(X) || Q(Y)`
- Programmer can force sequentiality
  - `P(X) then Q(Y)`
- Default is run in parallel if no dependences
  - `A := P(X); B := Q(Y)` -- can run in parallel
  - `Y := P(X); B := Q(Y)` -- cannot run in parallel

More Examples of ParaSail Parallelism

```
for X => Root then X.Left || X.Right while X not null
  concurrent loop
    Process(X.Data);   // Process called on each node in parallel
end loop;

concurrent interface Box<Element is Assignable<>> is
    func Create() -> Box;  // Creates an empty box
    func Put(locked var B : Box; E : Element);
    func Get(queued var B : Box) -> Element; // May wait
    func Get_Now(locked B : Box) -> optional Element;
end interface Box;

type Item_Box is Box<Item>;
var My_Box : Item_Box := Create();
```

Synchronizing ParaSail Parallelism

```
concurrent class Box <Element is Assignable<>> is
    var Content : optional Element; // starts out null
  exports
    func Create() -> Box is  // Creates an empty box
      return (Content => null);
    end func Create;

    func Put(locked var B : Box; E : Element) is
      B.Content := E;
    end func Put;

    func Get(queued var B : Box) -> Element is // May wait
     queued until B.Content not null then
      const Result := B.Content;
      B.Content := null;
      return Result;
    end func Get;

    func Get_Now(locked B : Box) -> optional Element is
      return B.Content;
    end func Get_Now;
end class Box;
```

ParaSail Virtual Machine

- ParaSail Virtual Machine (PSVM) designed for prototype implementations of ParaSail.
- PSVM designed to support “pico” threading with parallel block, parallel call, and parallel wait instructions.
- Heavier-weight “server” threads serve a queue of light-weight pico-threads, each of which represents a sequence of PSVM instructions (parallel block) or a single parallel “call”
  - Similar to Intel’s Cilk (and TBB) run-time model with *work stealing.*
- While waiting to be served, a pico-thread needs only a handful of words of memory.
- A single ParaSail program can easily involve 1000’s of pico threads.
- PSVM instrumented to show degree of parallelism achieved

Why and How to Formalize?

- Assertions help catch bugs sooner rather than later.
- Parallelism makes bugs much more expensive to find and fix.

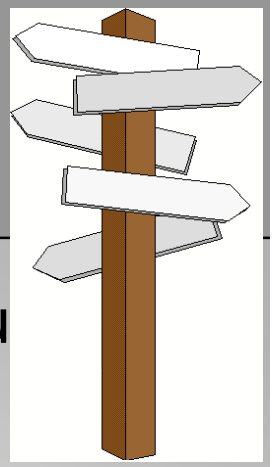
⇒ Integrate assertions (annotations) into the syntax everywhere, as pre/postconditions, invariants, etc.

⇒ Compiler disallows potential race-conditions.

⇒ Compiler checks assertions, rejects the program if it can't prove the assertions.

⇒ No run-time checking implies better performance, and no run-time exceptions to worry about.

Annotations in ParaSail



- Preconditions, Postconditions, Constraints, etc. all u
  same Hoare-like syntax: {X != 0}
- All assertions are checked at compile-time
  - no run-time checks inserted
  - no run-time exceptions to worry about or propagate
- Location of assertion determines whether is a:
  - precondition (before “->”)
  - postcondition (after “->”)
  - assertion (between statements)
  - constraint (in type definition)
  - invariant (at top-level of class definition)

Examples of ParaSail Annotations

```
interface Stack <Component is Assignable<>; Size_Type is Integer<>> is

    func Max_Stack_Size(S : Stack) -> Size_Type {Max_Stack_Size > 0};

    func Count(S : Stack) -> Size_Type
      {Count <= Max_Stack_Size(S)};

    func Create(Max : Size_Type {Max > 0}) -> Stack
      {Max_Stack_Size(Create) == Max; Count(Create) == 0};

    func Is_Empty(S : Stack) -> Boolean
      {Is_Empty == (Count(S) == 0)};

    func Is_Full(S : Stack) -> Boolean
      {Is_Full == (Count(S) == Max_Stack_Size(S))};

    func Push(var S : Stack {not Is_Full(S)}; X : Component)
      {Count(S') == Count(S) + 1};

    func Top(ref S : Stack {not Is_Empty(S)}) -> ref Component;

    func Pop(var S : Stack {not Is_Empty(S)})
      {Count(S') == Count(S) - 1};

end interface Stack;
```

More on Stack Annotations

```
class Stack <Component is Assignable<>; Size_Type is Integer<>> is
    const Max_Len : Size_Type;
    var Cur_Len : Size_Type {Cur_Len in 0..Max_Len};
    type Index_Type is Size_Type {Index_Type in 1..Max_Len};
    var Data : Array<optional Component, Indexed_By => Index_Type>;
  exports
    {for all I in 1..Cur_Len => Data[I] not null}   // invariant for Top()
    ...
    func Count(S : Stack) -> Size_Type
      {Count <= Max_Stack_Size(S)} is
      return S.Cur_Len;
    end func Count;

    func Create(Max : Size_Type {Max > 0}) -> Stack
      {Max_Stack_Size(Create) == Max; Count(Create) == 0} is
      return (Max_Len => Max, Cur_Len => 0, Data => [.. => null]);
    end func Create;

    func Push(var S : Stack {not Is_Full(S)}; X : Component)
      {Count(S') == Count(S) + 1} is
      S.Cur_Len += 1;            // requires not Is_Full(S) precondition
      S.Data[S.Cur_Len] := X;    // preserves invariant (see above)
    end func Push;

    func Top(ref S : Stack {not Is_Empty(S)}) -> ref Component is
      return S.Data[S.Cur_Len];// requires invariant (above) and not Is_Empty
    end func Top;
end class Stack;
```

More Annotation Examples

```
type Age is new Integer<0 .. 200>;  // a distinct integer type
type Youth is Age {Youth <= 20};    // a sub-range type
type Senior is Age {Senior >= 50};  // another sub-range type
-----------------------------------
func GCD(X, Y : Integer {X > 0; Y > 0}) -> Integer
   {GCD > 0; GCD <= X; GCD <= Y;
    X mod GCD == 0; Y mod GCD == 0} is
  var Result := X;      {Result > 0; X mod Result == 0}
  var Next := Y mod X; {Next <= Y; Y - Next mod Result == 0}

  while Next != 0 loop
    {Next > 0; Next < Result; Result <= X}
    const Old_Result := Result;
    Result := Next; {Result < Old_Result}
    Next := Old_Result mod Result;
    {Result > 0; Result <= Y; Old_Result - Next mod Result == 0}
  end loop;

  return Result;
end func GCD;
```

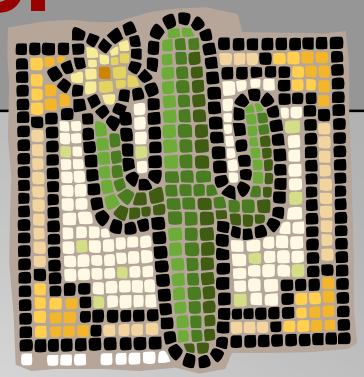
More on ParaSail Annotations

- Can declare annotation-only components and operations inside the "{ ... }"
  - Useful for pseudo-attributes like "taintedness" and states like "properly_initialized".
- All checking is at compile-time; no run-time exceptions
  - Exceptions don't play well when lots of threads running about
  - ParaSail *does* allow a block, loop, or operation to be "abruptly" exited with all but one thread killed off in the process.
    - Can be used by a monitoring thread to terminate a block and initiate some kind of recovery (perhaps due to resource exhaustion):

```
block
   Monitor(Resource); exit block with Result => null;
|| Do_Work(Resource, Data);
   exit block with Result => Data;
end block;
```

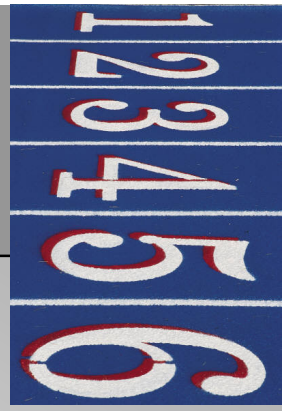
More Parasail Examples

Walk Parse Tree in Parallel



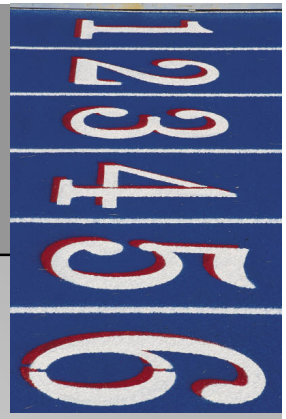
```
type Node_Kind is Enum < [#leaf, #unary, #binary] >;
  ...
for X => Root while X not null loop
  case X.Kind of
    [#leaf] =>
      Process_Leaf(X);
    [#unary] =>
      Process_Unary(X.Data) ||
      continue loop with X => X.Operand;
    [#binary] =>
      Process_Binary(X.Data) ||
      continue loop with X => X.Left ||
      continue loop with X => X.Right;
  end case;
end loop;
```

Map-Reduce in ParaSail



- Map-Reduce
  - defined as applying an operation to each element of a Vector *(map),* and then combining the results *(reduce)*
  - func Map(Input : Input_Type) → Output_Type
  - func Reduce(Left, Right : Output_Type) → Output_Type
  - func Map_Reduce(Vector<Input_Type>) → Output_Type

Map-Reduce code




```
func Map_Reduce
  (func Map(Input is Any<>) → (Output is Any<>);
   func Reduce(Left, Right : Output) → Output;
   Inputs : Vector<Input>)  {Length(Inputs) > 0} → Output is
      // Handle singleton directly, recurse for longer inputs
      if Length(Inputs) == 1 then
        return Map(Inputs[1]);
      else
        // Split and recurse
        const Half_Length := Length(Inputs)/2;
        return Reduce
          (Map_Reduce(Map, Reduce, Inputs[1..Half_Length]),
            Map_Reduce(Map, Reduce, Inputs[Half_Length <.. Length(Inputs)]));
      end if;
end func Map_Reduce;

func Test() is  // Test Map_Reduce function -- compute sum of squares
    Print_Int(Map_Reduce
      (Map => lambda(X : Integer) → Integer is (X**2),
       Reduce => "+",
       Inputs => [1, 2, 3, 4, 5, 6]));
end func Test;
```

Real-Time Programming: Queued locking used for Delay

```
abstract concurrent interface Clock <Time_Type is Ordered<>> is
   func Now(C : Clock) -> Time_Type;
   func Delay_Until(queued C : Clock; Wakeup : Time_Type)
      {Now(C') >= Wakeup};  // queued until Now(C) >= Wakeup
end interface Clock;

concurrent interface Real_Time_Clock<...> extends Clock<...> is
   func Create(...) -> Real_Time_Clock;
   ...
end interface Real_Time_Clock;

var My_Clock : Real_Time_Clock <...> := Create(...);
const Too_Late := Now(My_Clock) + Max_Wait;
block
   Delay_Until(My_Clock, Wakeup => Too_Late);
   exit block with (Result => null);
||
   const Data := Get(My_Box);
   Process(Data); exit block with (Result => Data);
end block;
```

Parallel N-Queens Interface

```
interface N_Queens <N : Univ_Integer := 8> is
   // Place N queens on an NxN checkerboard so that none of them can
   // "take" each other.  Return vector of solutions, each solution being
   // an array of columns indexed by row indicating placement of queens.
  type Chess_Unit is new Integer<-N*2 .. N*2>;
  type Row is Chess_Unit {Row in 1..N};
  type Column is Chess_Unit {Column in 1..N};
  type Solution is Array<optional Column, Indexed_By => Row>;

  func Place_Queens() -> Vector<Solution>
   {for all S of Place_Queens => for all Col of S => Col not null};
   // Produce a vector of solutions, with the requirement
   // that for each solution, there are non-null column numbers
   // specified for each row of the checkerboard.
end interface N_Queens;
```

Parallel N-Queens Class *(cont'd)*

```
class N_Queens is
    interface Solution_State<> is ... // local nested module
    class Solution_State is
       type Sum is Set<2..2*N>;                  // Diagonals where R+C = K
       type Diff is Set<(1-N) .. (N-1)>;         // Diagonals where R-C = K
       ...
    end class Solution_State;
  exports
    func Place_Queens() -> Vector<Solution>
     {for all S of Place_Queens => for all Col of S => Col not null}
    is
      var Solutions : concurrent Vector<Solution> := [];
     *Outer_Loop*
      for State : Solution_State := Initial_State() loop
          // Iterate over the columns
          ... // All done, remember trial result with last queen placed
          Solutions |= Final_Result(State, R);
          ...
      end loop Outer_Loop;
      return Solutions;
    end func Place_Queens;
end class N_Queens;
```

Parallel N-Queens Class *(cont'd)*

```
func Place_Queens() -> Vector<Solution> is
  var Solutions : concurrent Vector<Solution> := [];
 *Outer_Loop*
  for State : Solution_State := Initial_State() loop // over the columns
    for R in Row concurrent loop // over the rows (in parallel)
      if Is_Acceptable(State, R) then
        // Found a Row/Column combination that is not on any "busy" diagonal
        if Current_Column(State) < N then
                                   // Keep going since haven't reached Nth column.
          continue loop Outer_Loop with Next_State(State, R);
        else                       // All done, remember trial result with last queen placed
          Solutions |= Final_Result(State, R);
        end if;
      end if;
    end loop;
  end loop Outer_Loop;
  return Solutions;
end func Place_Queens;
```

Assessing Parasail

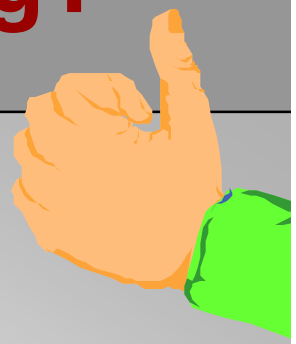
Summarizing ParaSail Model

- ParaSail has four basic concepts:
  - Module
    - has an Interface, and Classes that implement it
    - **interface** M <Formal **is** Int<>> **is ...**
    - Supports *inheritance* of interface and code
  - Type
    - is an instance of a Module
    - **type** T **is** M <Actual>;
    - "T+" is polymorphic type for types inheriting from T's interface
  - Object
    - is an instance of a Type
    - **var** Obj : T := Create(...);
  - Operation
    - is defined in a Module, and
    - operates on one or more Objects of specified Types.

User-defined Indexing, Literals, etc.

- User-defined indexing
  - Any type with **op** “indexing” defined
  - Indexing function returns **ref** to component of parameter
  - Built-in support for extensible structures, optional elements
- User-defined literals
  - Any type with **op** “from_univ” defined from:
    - Univ_Integer (42), Univ_Real (3.141592653589793)
    - Univ_String (“Hitchhiker’s Guide”), Univ_Character (‘π’)
    - Univ_Enumeration (#red)
- User-defined ordering
  - Define single binary **op** “=?” (pronounced “*compare*”)
  - Returns #less, #equal, #greater, #unordered
  - Implies “<=“, “<“, “==“, “!=“, “>”, “>=“, “in X..Y”, “not in X..Y”

What makes ParaSail Interesting?



- Pervasive (implicit and explicit) parallelism
  - Supported by ParaSail Virtual Machine
- Inherently safe:
  - preconditions, postconditions, constraints, etc., integrated throughout the syntax
  - no global variables; no dangling references; value semantics
  - no run-time checks or exceptions -- all checking at compile-time
  - storage management based on optional and extensible objects
- Small number of flexible concepts:
  - Modules, Types, Objects, Operations
- User-defined literals, indexing, aggregates, physical units checking
- It's got a cool name

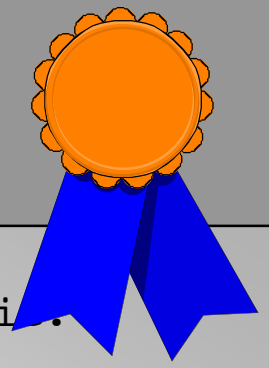
How does ParaSail Compare to ...

- C/C++ -- built-in safety; built-in parallelism; much simpler
- Ada -- eliminates race conditions, increases parallelism, eliminates run-time checks, simplifies language
- Java -- eliminates race conditions, increases parallelism, avoids garbage collection, does all checking at compile-time, no run-time exceptions
- OCaml/F# -- unifies modules and objects, eliminates exceptions, avoids garbage collection, increases parallelism, more emphasis on readability

Some of the Open Issues in ParaSail

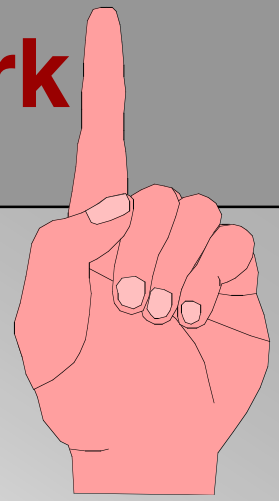
- Use of “Context” parameter for environment
  - such as “the” database or “the” user or “the” filesystem
- How to standardize how “smart” compiler is at proving assertions
  - Open source algorithm?
  - Detailed specification of inference and simplification rules?
  - A second-class assertion that need not be provable at compile-time?
  - Proof-carrying code? (easier to check than to prove)
- Open vs. Closed World Assumption for Postconditions
  - Which properties are presumed to be True after an operation if they were known True before the operation? What is the “frame”?
  - Is anything that is not provably True considered False or unknown?

Ultimate Test of Type Model: Physical Units Example



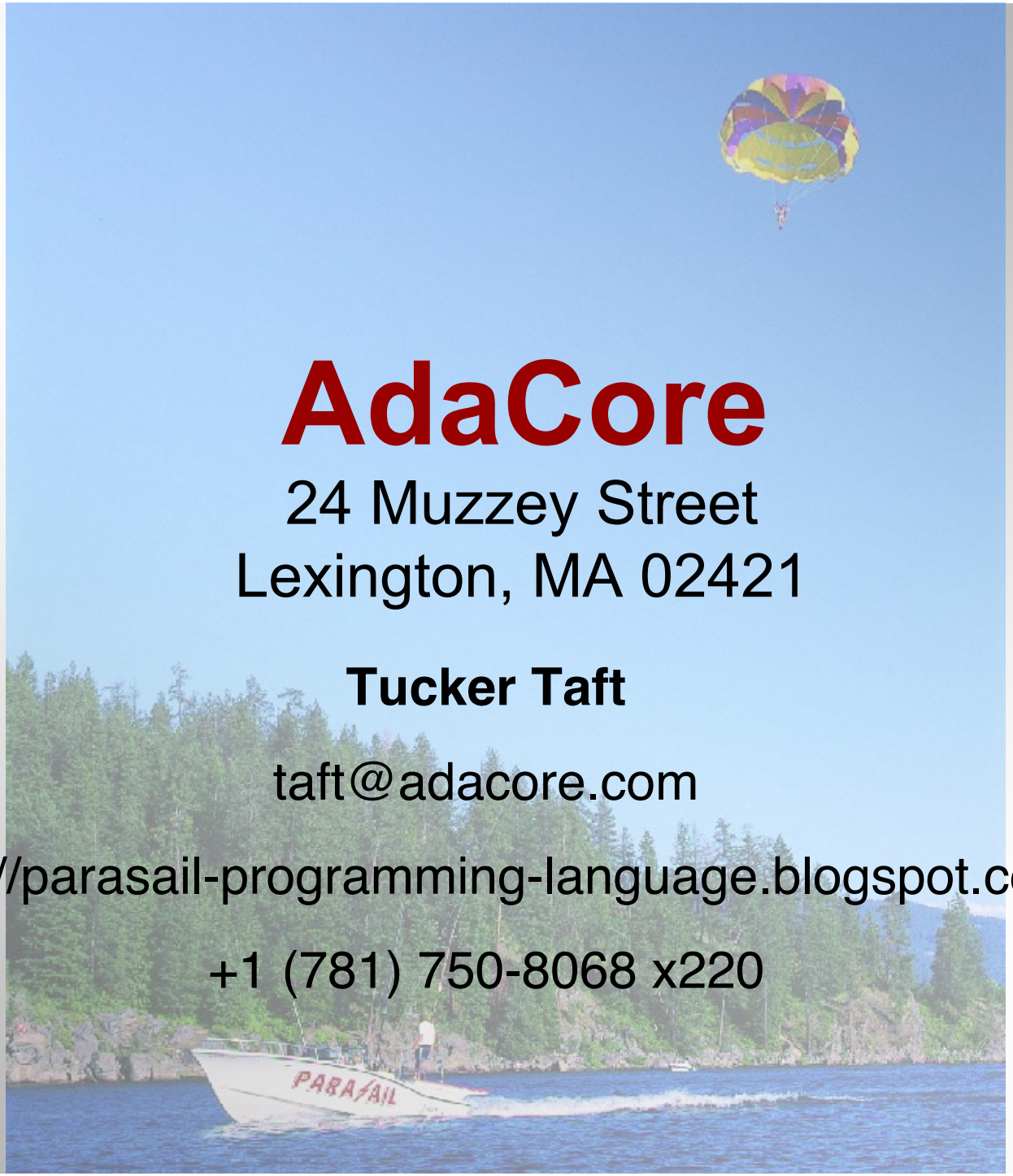
```
interface Float_With_Units<Dimension : Dimension_Vector> is
  // Real value with vector of exponents specifying the MKS units
  func "from_univ"(Univ: Univ_Real) -> Float_With_Units;
  op "+"(Left, Right: Float_With_Units) -> Float_With_Units;
  op "-"(Left, Right: Float_With_Units) -> Float_With_Units;
  op "+"(Right: Float_With_Units) -> Float_With_Units;
  op "-"(Right: Float_With_Units) -> Float_With_Units;
  op "*"(Left: Float_With_Units; Right: Right_Type is Float_With_Units<>)
    -> (Result: Result_Type is Float_With_Units
                                <Dimension + Right_Type::Dimension>);
  op "/"(Left: Float_With_Units; Right: Right_Type is Float_With_Units<>)
    -> (Result: Result_Type is Float_With_Units
                                <Dimension - Right_Type::Dimension>);
  op "**"(Left: Float_With_Units; <Right: Univ_Integer>)
    -> (Result: Result_Type is Float_With_Units<Dimension*Right>);
  func To_String(Val: Float_With_Units) -> Univ_String;
  func From_String(Str : Univ_String) -> Float_With_Units;
end interface Float_With_Units;

type Meters is Float_With_Units
    <Dimension => [#m => 1.0, #k => 0.0, #s => 0.0] >;
```

Conclusions and Ongoing Work



- It is productive to start from scratch now and then
- Can simplify and unify
- Can focus on new issues
  - pervasive parallelism
  - integrated annotations enforced at compile-time
- Ongoing work
  - Completing Prototype Compiler and ParaSail Virtual machine
  - Refining language from experience and feedback
- Read the blog if you are interested...

http://parasail-programming-language.blogspot.com



AdaCore

24 Muzzey Street
Lexington, MA 02421

Tucker Taft

taft@adacore.com

http://parasail-programming-language.blogspot.com

+1 (781) 750-8068 x220