ParaSail: Designing a safe, pervasively parallel language

The GNAT Pro Company

Tucker Taft
AdaCore Inc

February 2014

www.adacore.com

www.parasail-lang.org

Our Goal: Safe, Simple, Pervasively Parallel Programming

- **Why this goal?**
  - Deal with the unstoppable shift to multicore, manycore, GPGPU, and/or cloud computing.
- **Restated Goal: Make it easier and more natural to write parallel programs than sequential programs.**
- **… and oh by the way, what do we mean by "*parallel*" programming as opposed to "*concurrent*" programming?**
  - "*concurrent*" programming constructs allow programmer to *simplify* by using multiple threads to reflect the natural concurrency in the problem domain – heavier weight constructs OK
  - "*parallel*" programming constructs allow a programmer to *divide and conquer* a problem, using multiple threads to work in parallel on independent parts of the problem – constructs need to be light weight both syntactically and at run-time

The ParaSail experiment in simplified parallel programming

- **Eliminate global variables**
  - Operation can only access or update variable state via its parameters
- **Eliminate parameter aliasing**
  - Use "hand-off" semantics
- **Eliminate explicit threads, lock/unlock, signal/wait**
  - Concurrent objects synchronized automatically
- **Eliminate run-time exception handling**
  - Compile-time checking and propagation of preconditions
- **Eliminate pointers**
  - Adopt notion of "optional" objects that can grow and shrink
- **Eliminate global heap with no explicit allocate/free of storage and no garbage collector**
  - Replaced by region-based storage management (local heaps)
  - All objects conceptually live in a local stack frame

AdaCore
The GNAT Pro Company

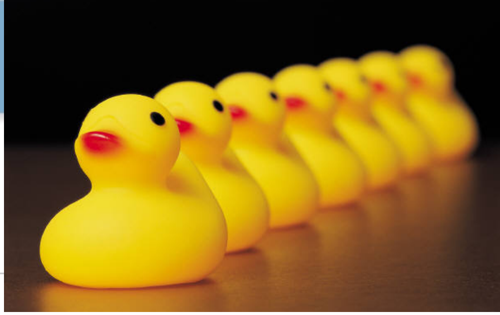
What ParaSail has left

- **Pervasive parallelism**
  - Parallel by default; it is *easier* to write in parallel than sequentially
  - *All* ParaSail expressions can be evaluated in parallel
    - In expression like "G(X) + H(Y)", G(X) and H(Y) can be evaluated in parallel
    - Applies to *recursive* calls as well (as in Word_Count example)
  - Statement executions can be interleaved if no data dependencies unless separated by explicit **then** rather than ";"
  - Loop iterations are *unordered* and possibly concurrent unless explicit **forward** or **reverse** is specified
  - Programmer can express *explicit* parallelism easily using "||" as statement connector, or **concurrent** on loop statement
    - Compiler will complain if any possible data dependencies
- **Full object-oriented programming model**
  - Full class-and-interface-based object-oriented programming
  - All modules are generic, but with fully shared compilation model
  - Convenient region-based automatic storage management
- **Annotations part of the syntax**
  - pre- and postconditions
  - class invariants and value predicates

ParaSail uses Syntactic Sugar to provide extensibility

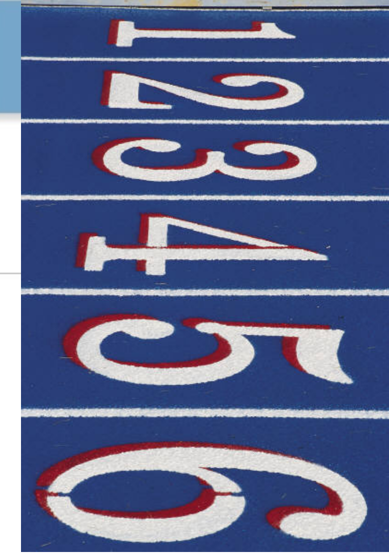
- **User-defined indexing**
  - Any type with **op** "indexing" defined
  - Indexing function returns **ref** to component of parameter
  - Built-in support for extensible structures, optional elements
- **User-defined literals**
  - Any type with **op** "from_univ" defined from:
    - Univ_Integer (42), 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"

Powerful Parallel Iterators – While loops, tail recursion, and backtracking considered sequential



How do *Iterators* Fit into this Picture?

- ***Computationally-intensive*** **Programs Typically Build, Analyze, Search, Summarize, and/or Transform** ***Large Data Structures*** **or** ***Large Data Spaces***
- ***Iterators*** **encapsulate the process of walking data structures or data spaces**
- **The biggest** ***speed-up*** **from parallelism is provided by** ***spreading*** **the processing of a large data structure or data space across multiple processing units**
- **So…high level iterators that are** ***amenable*** **to a** ***safe, parallel interpretation*** **can be critical to capitalizing on distributed and/or multicore hardware.**



Simple Iterator and Sequential Equivalents

```
for I in 1..N loop
   P(I)
end loop

I := 1; while I <= N loop
   P(I)
   I += 1   // compute next value of I
end loop

let LP = lambda (I)
   if I <= N then
      P(I); LP(I+1)  // tail recursion
   end if
in LP(1)
```

Linked-list Iterator with Sequential Equivalents

```
for (node *p = first; p; p = p->next) {
    process(p)
}

node *p = first; while(p) {
    process(p)
    p = p->next  // point to next node
}

let lp = lambda (node *p) {
    if (p) {
        process(p)
        lp(p->next)  // tail recursion
    }
} in lp(first)
```

The Trouble with *While* loops and Tail recursion

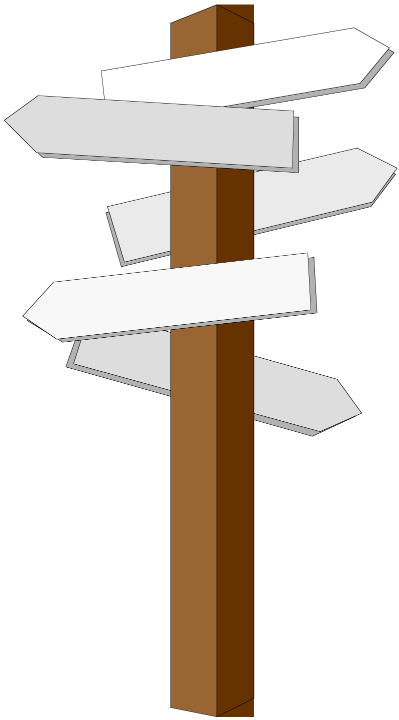
+ **While loop – pros:**
  – Universal sequential loop construct; semantics defined simply
– **While loop – cons:**
  – Necessarily updates a global to advance through iteration
  – Generally doesn't update global until *after* finishing processing current iteration
+ **Tail recursion – pros:**
  – No need for global variables – each loop iteration carries its own copy of loop variable(s)
  – Can generalize to walking more complex data structure such as a tree by recursing on multiple subtrees
– **Tail recursion – cons:**
  – Next iteration value not specified until making (tail) recursive call
  – Each loop necessarily becomes a separate function

Combine "pros" of Tail Recursion with (parallel) "for" loop

- **Parallelism requires each iteration to carry its *own copy* of loop variable(s), like tail recursion**
  - For-loop variable treated as local constant of each loop iteration
- **For loop syntax allows next iteration value to be specified *before* beginning current iteration**
  - rather than at tail-recursion point or end of loop body
  - multiple iterations can be initiated in parallel
- **Explicit "continue" statement may be used to handle more complex iteration requirements**
  - condition can determine loop-variable values for next iteration(s)
- **Explicit "parallel" statement connector allows "continue" statement to be executed in parallel with current iteration**
  - rather than *after* the current iteration is complete
- **Explicit "exit" or "return" allows easy premature exit**

Walking a tree structure with explicit "continue" statements

```
for P => Root while P not null loop
 case P.Kind of
  [#binary] =>
       continue loop with P.Left
    || continue loop with P.Right
    || Process_Binary (P.Data)
  [#unary] =>
       continue loop with P.Operand
    || Process_Unary (P.Data)
  [#leaf] =>
       Process_Leaf (P)
 end case
end loop
```



AdaCore
The GNAT Pro Company

Continue statement creates a new iteration – loop is effectively a "bag" of iterations (like a work list)

- **Each continue statement in this kind of explicit "value" iterator starts another iteration**
  - Iteration is added to "bag" of iterations associated with loop
  - Completes current thread of control
- **Can actually "continue" an *outer* loop**
  - Starts a new iteration of the enclosing loop – i.e. adds a new iteration to "bag" of outer loop – like a work-list
  - Inner loop keeps executing until all of the iterations already in its "bag" (work-list) are complete

```
*Outer_Loop*
 for Board : Chess_Board_State := No_Columns() loop     // N-Queens
    for R in 1 .. N concurrent loop
        if Can_Add_Queen(Board, R) then
           if Num_Columns(Board) < N then
              continue loop Outer_Loop with Board => Add_Column(Board, R)
           else
              Solutions |= Add_Column(Board, R)
```

Short-hand when Simple Binary Tree Iteration

```
for P => Root then P.Left || P.Right
   while P not null concurrent loop
      // "concurrent" means next iteration(s) start immed.
      Process (P.Data)
end loop

// Iterator in a quantified expression, e.g.
// "at least one node has a positive count"
(for some P => Root then P.Left || P.Right
    while P not null => P.Count > 0)

// Or flatten binary tree using a vector comprehension:
[ for P => Root then P.Left || P.Right
    while P not null => P.Data ]
```

Generalize combining construct to provide Map/Reduce

- Expression in <…> gives *initial* value, and is replaced after each computation with result
- *Associativity* of operation allows parallelism
- Can be easier to comprehend than **foldl, foldr, foldl1, …**

```
// Compute sum of squares of counts
Sum_Sqrs :=
  (for P => Root then P.Left || P.Right
     while P not null => <0> + P.Count**2)
```

```
// Compute max of counts (Max(null, A) == A)
Max_Count :=
  (for P => Root then P.Left || P.Right
    while P not null => Max(<null>, P.Count))
```

Two kinds of user-defined "container" iterators

- **User-defined "set" iterator**
  - **for** I **in** <set> **loop** …
  - "Set" abstraction must provide "Remove_Any" operation, and may provide "Remove_First" and "Remove_Last" for ordered (**forward** or **reverse)** iteration
  - Remove_... operation returns null when set is empty
  - *Copy* of set made and then Remove_... destructively empties the set
- **User-defined "map" iterator iterates over index set of map**
  - **for each** [K => V] **of** <map> **loop** …
  - "Map" abstraction must provide "Index_Set" and "Indexing" operations.
  - Index_Set returns set of keys of all (non-null) values in Map
  - Indexing returns ref to Value given Key and ref to Map
  - Short hand "**for each** V **of** <map> **loop** …" uses an anonymous variable for the Key

Examples of "set" and "map" iterators

- Set iterators:

```
for I in 1..10 forward loop …
for I in 1 | 3 | 5 | 7 reverse loop …
for S in Successors(G, N) concurrent loop …
```

- Map iterators:

```
for each [N => Node] of G loop …   // graph
for each [Name => Sym] of Sym_Tab loop …
for each Elem of Vec
                forward loop … // index implicit
```

- Compiler automatically generates code to:
  - Copy set (or call Index_Set); bind result of Remove_{First,Last,Any} to loop var/key until **null**
  - For map, bind V in [K=>V] to "Indexing(map, K)"

Map and Set primitives need *not* support concurrent access

- **Iterations are created by calling Remove_... sequentially**
  - once for each iteration until it returns null
- **Each iteration carries its own loop-var value(s)**
- **If "concurrent" loop, next call on Remove_...**
  - is performed before doing body of loop and
  - next iteration can then run in parallel
- **If non-concurrent loop**
  - will wait until body completes before creating next iteration.
- **Sequential use of Remove_... even when "concurrent"**
  - simplifies creating user-defined iterable abstractions
- **"Continue" for map/set iterators simply skips rest of body of loop**
  - creates next iteration if not already done.

Iterators can have filters

- Filter specified at end of iterator as boolean expr in {...}
- Only values where filter evaluates #true are included in iteration

```
func Qsort(V : Vector<Comparable>) -> Vector<Comparable>
is
  if |V| <= 1 then
    return V  // The easy case
  else
    const Pivot := V[ |V|/2 ]
    return Qsort([ for each E of V {E <= Pivot} => E ])
         | Qsort([ for each E of V {E > Pivot} => E ])
  end if
end func Qsort
```

Can "continue" outer loop, as in Breadth-First Search of Graph

```
var Seen : Array<Atomic<Boolean>, Node_Id> :=
                    [for N in G.All_Nodes() => Atomic(#false)]
*Outer*
  for Next_Set => Root_Node_Set loop // specify node-set to search
    for N in Next_Set { not Value(Seen[N]) } concurrent loop
      //  Check each node in node set, in parallel
      Set(Seen[N], #true)  // "benign" race condition
      if not Is_Target(G[N]) then
         // Start new iteration of outer loop with successor set
         continue loop Outer with Next_Set => G[N].Succs
      else
         // Found a node that satisfies Is_Target
         // This "return" will cancel other concurrent threads
         return N
      end if
    end loop
  end loop Outer
  // No node found that satisfies Is_Target(…)
  return null
```

Synchronization in ParaSail via Concurrent Objects

- **No aliasing, and no concurrent updates allowed when using “normal” ParaSail (sequential) objects**
- **What to do if multiple writers, or concurrent reading and writing is desired?**
  - Can *slice* large container – restrict access to subset of indices
  - Can create a **concurrent** object

```
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**

Example ParaSail Virtual Machine Statistics

Command to execute: stats

Region Statistics:

New allocations by owner: 7326 = 78%
Re-allocations by owner: 849 = 9%
Total allocations by owner: 8175 = 87%

New allocations by non-owner: 851 = 9%
Re-allocations by non-owner: 348 = 3%
Total allocations by non-owner: 1199 = 12%

Total allocations: 9374

Threading Statistics:

Num_Initial_Thread_Servers : 3 + 1
Num_Dynamically_Allocated_Thread_Servers : 0
Max_Waiting_Threads (on some server's queue): 25
Average waiting threads: 12.89
Max_Active (threads): 4
Average active threads: 3.76
Max_Active_Masters : 32
Max_Subthreads_Per_Master : 16
Max_Waiting_For_Subthreads : 29
Num_Thread_Steals : 210 out of 1097 total thread initiations = 19%

Supporting Formal Methods in the Language

Compile-Time **Exception Handling**

Why and How to Formalize?

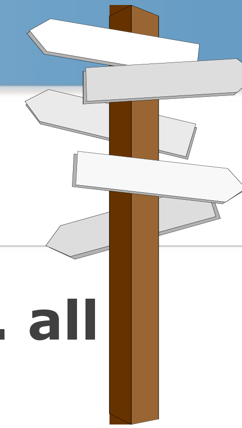
- *Assertions help catch bugs sooner rather than later.*
- *Parallelism makes bugs much more expensive to find and fix.*

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

⇒ **Compiler disallows potential race-conditions.**

⇒ **Compiler checks assertions, complains if it can't prove the assertions.**

⇒ *Substituting compile-time checking for run-time checking implies better performance, and allows problematic code to be identified earlier*



Annotations in ParaSail

- **Preconditions, Postconditions, Constraints, etc. all use Hoare-like syntax, such as "*{ X != 0 }*":**
  - ```
    func Pop(var S : Stack) {Count(S) > 0}
        -> Elem_Type {Count(S') == Count(S) - 1};
    ```
- **All assertions are checked at compile-time**
  - Preconditions can be used to help make assertions provable
  - Compile-time propagation to callers via preconditions

  => *Compile-time* exception handling
- **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 on ParaSail Annotations

- **Can declare annotation-only components and operations inside the "{ … }"**
  - Useful for pseudo-attributes like "taintedness" and states like "properly_initialized".
- **Checked at compile-time; no run-time exception handling**
  - 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;
```

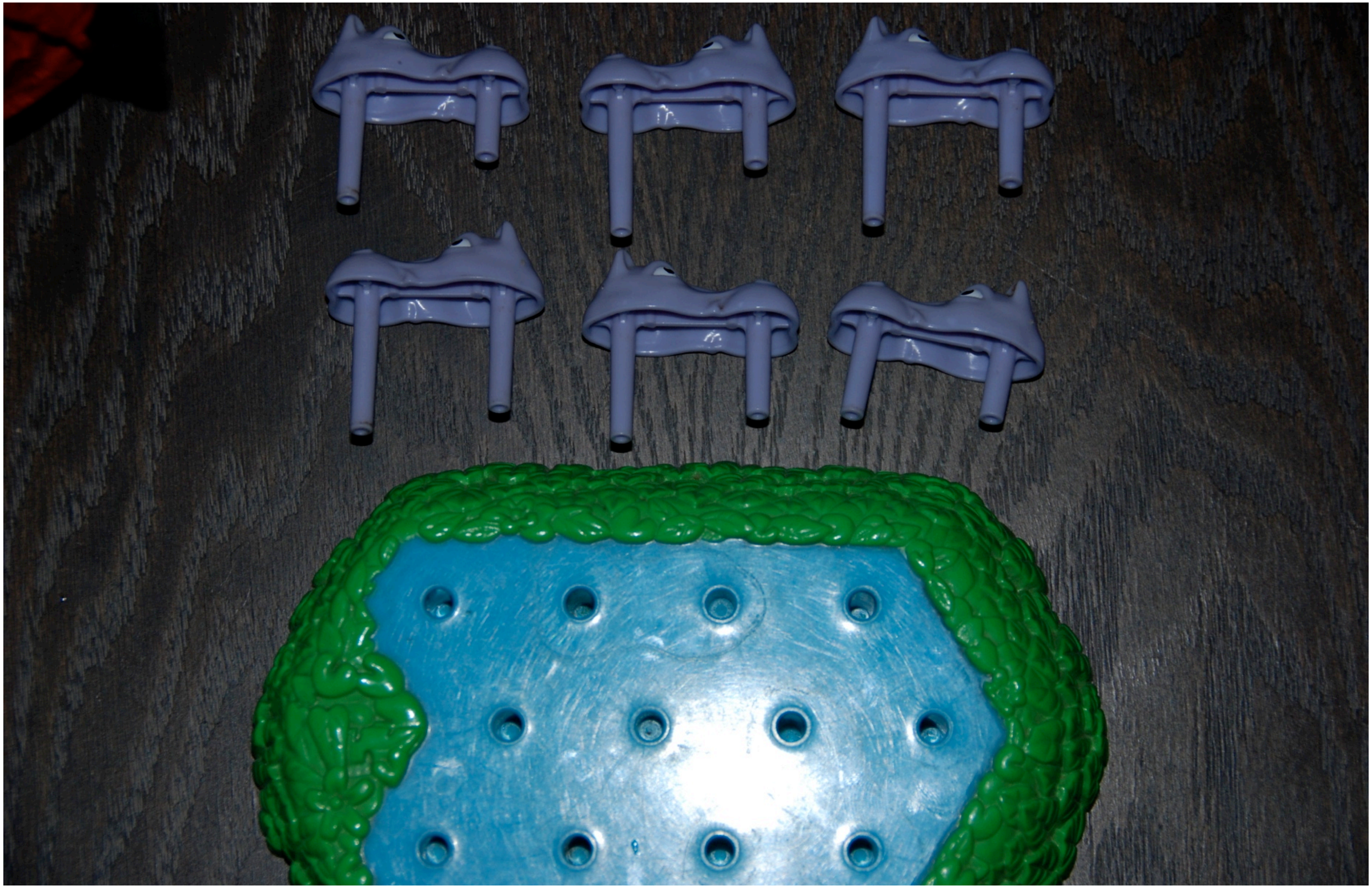
Multiply and Conquer

Searching a Game Tree

Hippo Game via "multiply"-and-conquer

- **Similar to N-Queens problem**
  - Place six hippos on game board so each sits flat
  - Each hippo has 2 posts, each of length 2 to 5
  - Board has 12 holes arranged in 3 rows of 4 each
    - Offset to produce equilateral triangles
  - On order of 6 factorial possible hippo arrangements
- **Rather than "divide-and-conquer" we use a sort of "multiply" and conquer**
  - Conceptually we keep creating more and more game boards, each with a partial solution
  - We hand out the partial solutions to multiple picothreads and have them try to solve rest of puzzle
  - Picothread places one more hippo, and then hands out to yet more picothreads to continue from there
  - First picothread to complete the puzzle kills off all of the others
  - Multiply and conquer means no explicit backtracking

Six Hippos, 12 posts, 12 holes



An incorrect solution

Hippo Game solution – Multiply and conquer (in Parython)

Termination condition

Multiply and Conquer

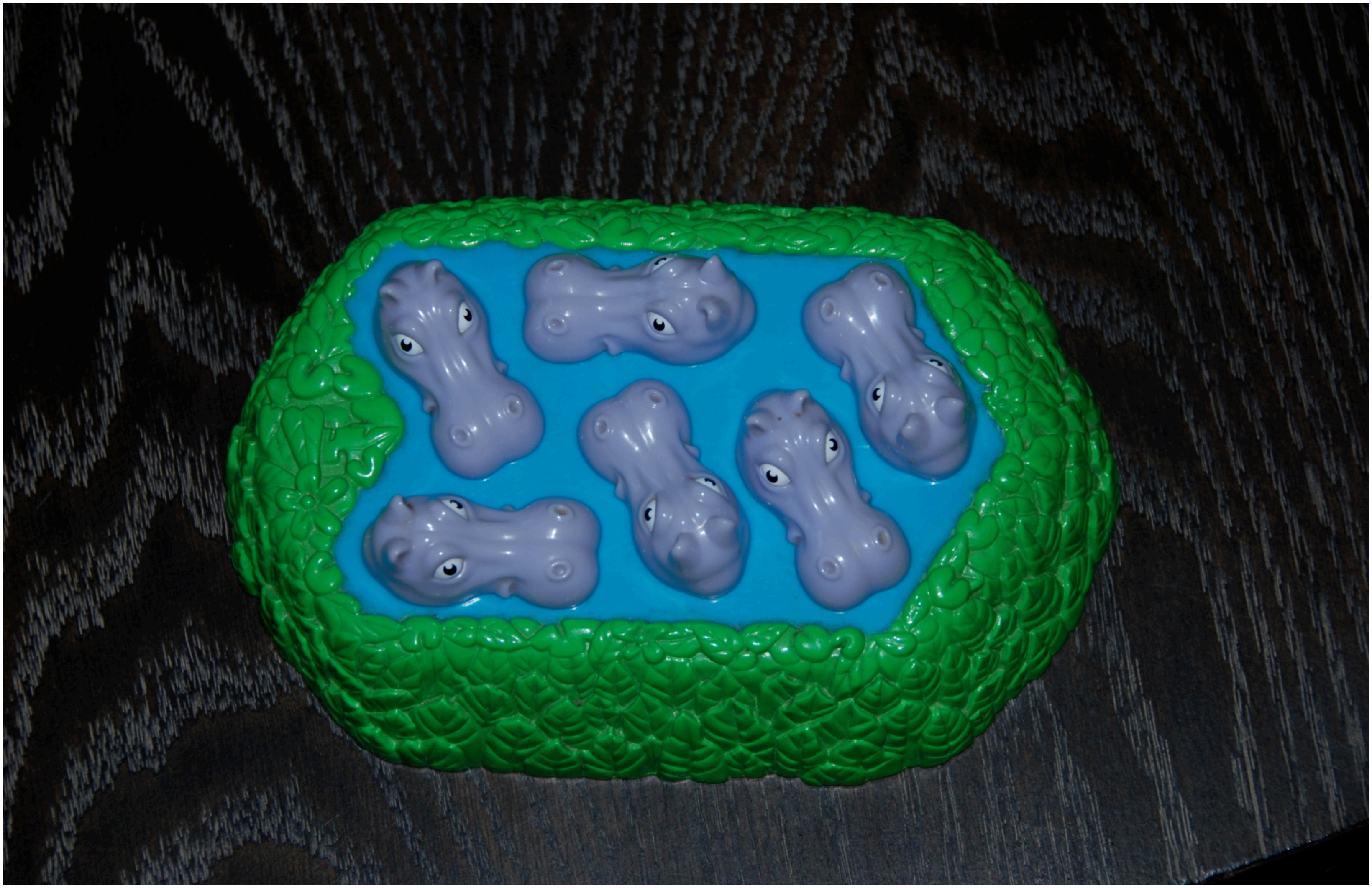
```
def Hippo_Game(Graph : Hole_Graph; Pieces : Vector<Hippo_Piece>)
  -> Game_Solution:
    *Outer*
     for (Index = 0;
       Open_Holes : Hole_Graph.Node_Set = All_Nodes(Graph);
       Partial_Solution : Game_Solution = []):

         if Index >= |Pieces|:
            return Partial_Solution  # found a complete solution

         const Piece = Pieces[Index]
         for Long_Loc in Open_Holes
          if Graph[Long_Loc].Depth >= Piece.Long concurrent:

            for Short_Loc in Successors(Graph, Long_Loc)
             if Short_Loc in Open_Holes
              and Graph[Short_Loc].Depth >= Piece.Short concurrent:
               # Found a pair of adjacent open holes that work
               # Add them into the solution we are building.
               const Next_Solution : Game_Solution =
                 Partial_Solution | {Index : (Long_Loc, Short_Loc)}
               # Continue the outer iteration with the next piece
               continue loop Outer with
                 (Index = Index + 1,
                  Open_Holes = Open_Holes – {Long_Loc, Short_Loc},
                  Partial_Solution = Next_Solution)
```

The (one and only) correct solution



Hippo Game Region and Work-Stealing Statistics

Command to execute: Place_Hippos
Piece 4,3 is at 11,21
Piece 5,2 is at 12,13
Piece 5,4 is at 14,24
Piece 4,2 is at 33,22
Piece 5,3 is at 23,34
Piece 3,2 is at 32,31
Command to execute: stats

Region Statistics:
New allocations by owner: 7326 = 78%
Re-allocations by owner: 849 = 9%
Total allocations by owner: 8175 = 87%

New allocations by non-owner: 851 = 9%
Re-allocations by non-owner: 348 = 3%
Total allocations by non-owner: 1199 = 12%

Total allocations: 9374

Threading Statistics:
Num_Initial_Thread_Servers : 3 + 1
Num_Dynamically_Allocated_Thread_Servers : 0
Max_Waiting_Threads (on some server's queue): 25
Average waiting threads: 12.89
Max_Active (threads): 4
Average active threads: 3.76
Max_Active_Masters : 32
Max_Subthreads_Per_Master : 16
Max_Waiting_For_Subthreads : 29
Num_Thread_Steals : 210 out of 1097 total thread initiations = 19%

Summary and Conclusions (and a plug for HILT 2014)

Summary of ParaSail Iterators

- **Flexible Iterators simplify walking a large Data Structure or Data Space**
  - Sequential equivalents such as while loops and tail recursion do not easily adapt to parallel interpretation
    - Next value not available until end of current iteration
    - Each iteration should carry its "own" loop-var values
- **Potential for significant speed-up can be enabled simply through use of higher-level iterators with safe parallel semantics**
  - Parallel versions of "continue," "exit," and "return" add to power and flexibility – bag of iterations model – can exit and cancel all others
- **Three kinds of iterators**
  - Explicit "value" iterator – Start, Next, While + continue – bag of threads
  - Set iterator – User provides Remove_{Any,First,Last} operation
  - Map iterator – User provides Index_Set and Indexing operations
- **Flexible Map/Reduce construct built using iterator**

Summary of ParaSail Annotations

- **Incorporating Annotations into syntax allows more complete specification of abstractions**
- **Preconditions can be used to propagate safety requirements to caller**
- **Compiler complaints about remaining possible run-time errors inidicates where more annotations are needed**
- **Effectively provides *compile-time* exception handling; remaining run-time problems handled otherwise:**
  - Null values can be used uniformly to signal *no result*
    - Only can happen if function result declared **optional**
  - Separate threads can be used to monitor for unpredictable error situations
    - Resource exhaustion
    - Time-outs

Conclusions

- Simple language can still be powerful
- Can eliminate features which create a sequential *bias*
- Can provide a language in which simple things are simple
- But can still build complex, parallel applications that are inherently safe

HILT 2014 – High Integrity Language Technology – Oct 20-21

- **Using *modeling*, *programming*, *verification*, … languages to support development, testing, and verification of *High Integrity* systems**
- ***Co-Located* with SPLASH/OOPSLA 2014 in October in Portland, OR, USA, October 20-24**
- **Join *Program Committee*, and *submit* papers**

HILT 2014

SPLASH
PORTLAND 2014