AdaCore
The GNAT Pro Company

ParaSail

Parallel Specification and Implementation Language

Less is More with Multicore

Tucker Taft
AdaCore Inc

February 2014

AdaCore
The GNAT Pro Company

Outline of Presentation

- **Why is the Hardware World moving to Multicore?**
  - And what does this mean for the Software World?
- **ParaSail: A simplified approach to safe parallel programming**
  - Pointer-free Divide-and-Conquer Parallel Programming
  - Region-Based Storage Management instead of garbage collection
  - Managing Parallelism using Work-Stealing
- **Conclusions**

Why is the hardware world moving to multi/manycore?

- **Power, power, power**
  - Increasing clock rates past 3GHz increased power density beyond what the chips (and customer pocketbooks) could bear.
  - More and more computing is moving to battery-operated mobile platforms where low power is king
- **With multi/manycore, the theoretical computing performance per watt can be increased by adding cores, and perhaps slowing clock rate a bit**
  - With single core, the performance per watt began to *decrease* with increasing clock rates, due to increased source-to-drain leakage.
- **Clock rate doubling all came to a screeching halt in about 2005**

The *Right Turn* in Single-Processor Performance

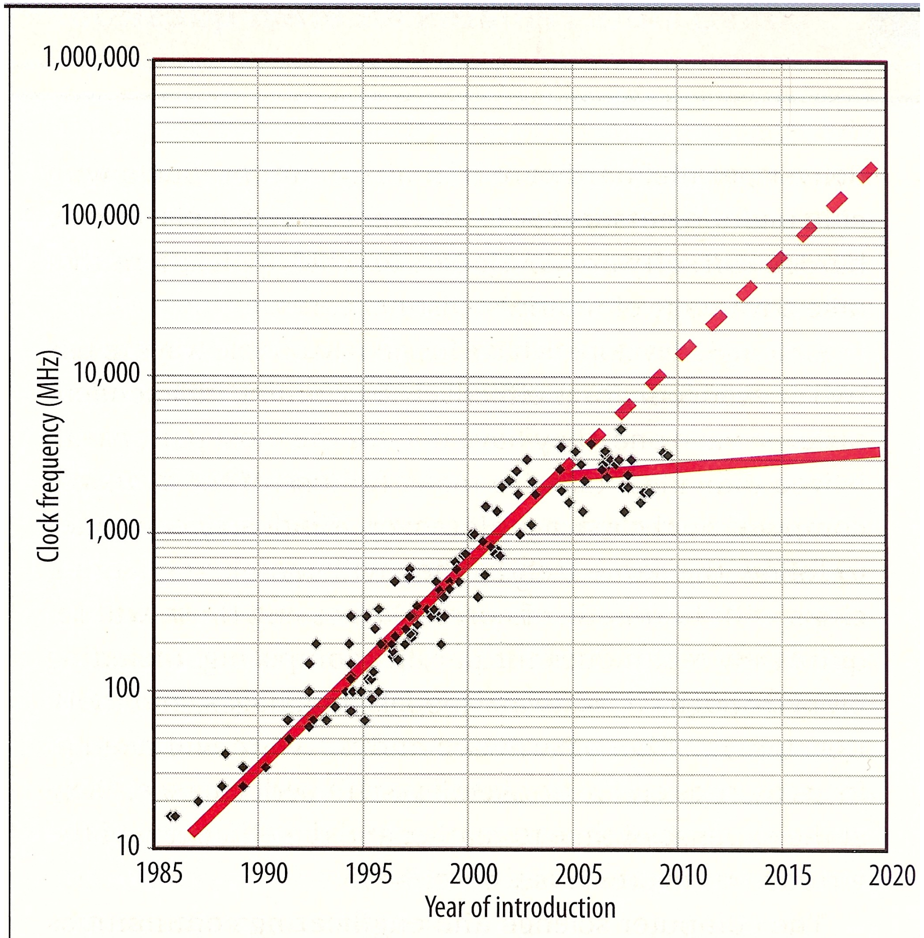


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy IEEE Computer, January 2011, page 33.

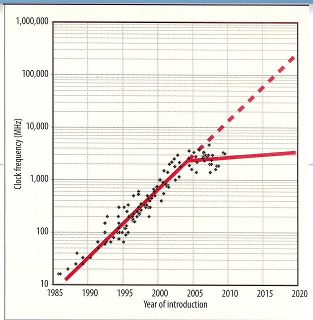


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

What are the implications of this Right Turn?

- **Clock rate implications**
  - Clock rates were doubling about every 2 years
  - Clock rates stalled at about 3Ghz in 2005
  - Had they continued doubling, we would now be buying laptops with clocks at about 50 Ghz.
- **Cores/chip implications**
  - Scaling to smaller features has continued
  - Now using added chip real estate for additional CPU “cores”
  - The number of cores/chip has started doubling since 2005
  - It has been 7+ years, and mainstream commercial x86 chips are now at 10 to 16 cores/chip, Xeon Phi at 50+, GPUs at 1000+
- **Back on Moore’s-Law exponential rocket**
  - But only if considering cores/chip x performance/core

ParaSail: *A simplified approach to safe and secure parallel programming*

Mutable Objects with Value Semantics

Stack-Based Heap Management

Compile-Time Exception Handling

Race-Free Parallel Programming

Why Design A New Parallel Language for Mission-Critical Programming?

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

Parallel programming languages can simplify multi/ manycore programming

- **As number of cores increases, traditional multithreading approaches become unwieldy**
  - Compiler ignoring availability of extra cores would be like a compiler ignoring availability of extra registers in a machine and forcing programmer to use them explicitly
  - Forcing programmer to worry about possible race conditions would be like requiring programmer to handle register allocation, or to worry about memory segmentation
- **Cores are a resource, like virtual memory or registers**
  - Compiler should be in charge of using cores wisely
  - Algorithm as expressed in programming language should allow compiler maximum freedom in using cores
  - Number of cores available should not affect difficulty of programmer's job or correctness of algorithm

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

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

Example: *Implicit* parallelism in ParaSail using divide-and-conquer

Simple cases

Implicitly Parallel Divide and Conquer

```
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 |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 := |S|/2
        const Sum := Word_Count( S[ 1 .. Half_Len ], Separators ) +
          Word_Count( S[ Half_Len <.. |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
```

AdaCore
The GNAT Pro Company

Overall ParaSail Model

- **ParaSail has four basic concepts:**
  - Module
    - has an Interface, and Classes that implement it
    - is always parametrized: **interface** M <Formal **is** Int<>> **is …**
    - supports *inheritance* of interface and code
  - Type
    - is an instance of a Module – specify module parameters
    - **type** T **is** [**new**] M <Actual>;
    - "T+" is polymorphic type for types implementing T's interface
  - Object
    - is an instance of a Type; is **var** or **const**
    - **var** Obj : T := Create(…);
  - Operation
    - is defined in a Module, and
    - operates on one or more Objects of specified Types.
    - are visible automatically based on types of parameters/result

AdaCore
The GNAT Pro Company

Why The Simplifications? Especially, why Pointer Free?

- **Consider F(X) + G(Y)**
  - We want to be able to safely evaluate F(X) and G(Y) in parallel *without* looking inside of F or G
  - Presume X and/or Y might be incoming **var** (in-out) parameters to the enclosing operation
  - No global variables is clearly pretty helpful
    - Otherwise F and G might be stepping on same object
  - No parameter aliasing is important, so we know X and Y do not refer to the same object
  - What do we do if X and Y are pointers?
    - Without more information, we must presume that from X and Y you could *reach* a common object Z
    - How do parameter modes (in-out vs. in, **var** vs. non-**var**) relate to objects accessible via pointers?

Result: pure *value semantics* for non-concurrent objects

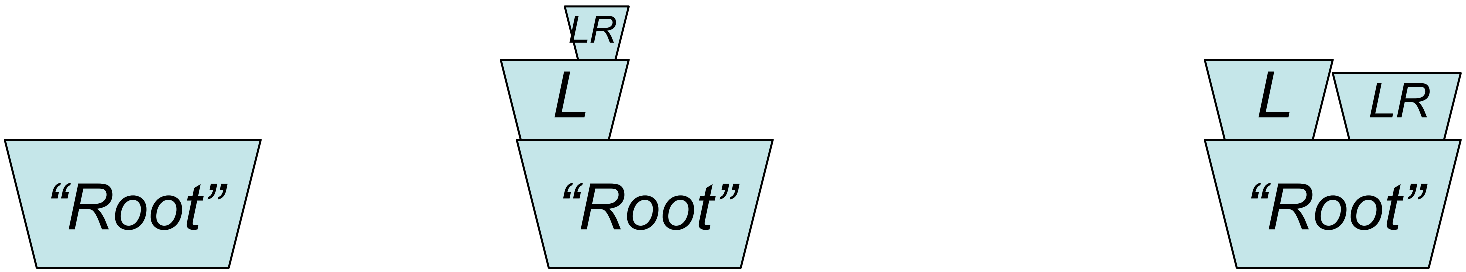
Expandable Containers Instead of Pointers

- **All types have additional null value; objects can be declared optional (i.e.null is OK) and can grow and shrink**
  - Eliminates many of the common uses for pointers, e.g. trees
  - Assignment (":=") is by copy
    - Move ("<==") and swap ("<=>") operators also provided
- **Generalized indexing into containers replaces pointers for cyclic structures**
  - for each N in Directed_Graph[I].Successors loop ...
- **Region-Based Storage Mgmt can replace Global Heap**
  - All objects are "local" with growth/shrinkage using local heap
  - "null" value carries indication of region to use on growth
- **Short-lived references to existing objects are permitted**
  - Returned by user-defined indexing functions, for example
  - Used to iterate over a data structure

Pointer-Free Trees

```
interface Tree_Node
   <Payload_Type is Assignable<>> is
    var Payload : Payload_Type;
    var Left : optional Tree_Node := null;
    var Right : optional Tree_Node := null;
end interface Tree_Node;

var Root : Tree_Node<Univ_String> := (Payload => "Root");
Root.Left := (Payload => "L", Right => (Payload => "LR"));
Root.Right <== Root.Left.Right;  // Root.Left.Right now null
```



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;
```

Other ParaSail Module/Type Features

- **Objects: `"var Obj:T;"` or `"const Obj: T := ..."`**
  - `Obj.Op(...)` is equivalent to `Op(Obj, ...)`
  - Compiler looks in all associated modules of operands for operation of given name; "T::Op" to specify location of Op
  - Operators like `"+"` treated uniformly, `Obj + X` is equivalent to `"+"(Obj, X)` and `T::"+"(Obj, X)` and `Obj."+"(X)`
- **User-defined literals: 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 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 "|=" and then compiler will create temps to support "X | Y" and "[...]" aggregates.

More Examples of ParaSail Parallelism and Synchronization

```
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**

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%

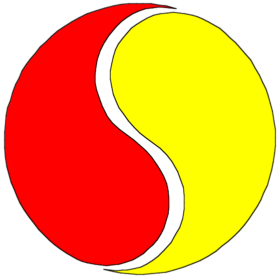
Summary of ParaSail extensibility

- **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”

Conclusions

Conclusions

- **Multicore Era is here**
  - Staying on Moore's Law "rocket" depends on using multiple cores
  - New languages supporting various parallel programming paradigms
  - Some languages moving toward implicit parallelism,
    - Compiler and run-time support using cores as resources, much as they have used registers and virtual memory
- **Simplified Language can enable Parallel-by-default programming**
  - *Mutable Objects with* Value Semantics
  - *Stack-Based* Heap Management
  - *Compile-Time* Exception Handling
  - *Race-Free* Parallel Programming



- **Parallel programming can be productive, safe, and enjoyable**
  - Can eliminate the sequential biases of existing languages
  - Can preserve a familiar Class-and-Interface-based Model
  - Can discover interesting new parallel programming idioms
- **Blog: http://parasail-programming-language.blogspot.com**