AdaCore
The GNAT Pro Company

Safe, Contract-Based Parallel Programming

Ada-Europe 2017
June 2017
Vienna, Austria

Tucker Taft
AdaCore Inc

Outline

- **Vocabulary for Parallel Programming**
- **Approaches to Supporting Parallel Programming**
  - With examples from Ada 202X, Rust, ParaSail, Go, etc.
- **Fostering a Parallel Programming Mindset**
- **Enforcing Safety in Parallel Programming**
- **Additional Kinds of Contracts for Parallel Programming**

Vocabulary

- Concurrency vs. Parallelism
- Program, Processor, Process
- Thread, Task, Job
- Strand, Picothread, Tasklet, Lightweight Task, Work Item (cf. Workqueue Algorithms)
- Server, Worker, Executor, Execution Agent, Kernel/OS Thread, Virtual CPU

Vocabulary
What is Concurrency vs. Parallelism?

Concurrency

- "***concurrent***" programming constructs allow programmer to …

Parallelism

- "***parallel***" programming constructs allow programmer to …

Concurrency vs. Parallelism

Concurrency

- "***concurrent***" programming constructs allow programmer to *simplify the program* by using multiple logical threads of control to reflect the natural concurrency in the problem domain
  - heavier weight constructs OK

Parallelism

- "***parallel***" programming constructs allow programmer to *divide and conquer* a problem, using multiple threads to work in parallel on independent parts of the problem to achieve a *net speedup*
  - constructs need to be light weight both syntactically and at run-time

Threads, Picothreads, Tasks, Tasklets, etc.

- **No uniformity in naming of threads of control within a process**
  - Thread, *Kernel Thread*, OS Thread, Task, Job, Light-Weight Process, Virtual CPU, Virtual Processor, Execution Agent, Executor, *Server Thread*, Worker Thread
  - "Task" generally describes a logical piece of work
  - "Thread" generally describes a virtual CPU, a thread of control within a process
  - "Job" in the context of a real-time system generally describes a single period's actions within a periodic task
- **No uniformity in naming of user-level very lightweight threads**
  - Task, Microthread, *Picothread*, Strand, *Tasklet*, Fiber, Lightweight Thread, *Work Item*
  - "User level" -- scheduling is done by code outside of the kernel/ operating-system

Vocabulary we will use

- **Focus on Parallelism *within* context of Concurrency**
- **“Task” will be used as a higher-level logical piece of work to do**
  - Task can potentially use parallelism internally
- **“Picothread,” “Tasklet,” “Work Item” will be used to talk about smaller unit of work to be performed sequentially**
- **“Kernel Thread” or “Server (Thread)” will be used to talk about a resource for executing picothreads**

Various Approaches to Supporting Parallel Programming

- **Library Approach**
  - Provide an API for spawning and waiting for tasklets
  - Some sort of synchronization – at least “mutex”es
  - Examples include TBB, Java Fork/Join, Rust
- **Pragma Approach**
  - No new syntax
  - Everything controlled by pragmas on:
    - Declarations
    - Loops
    - Blocks
  - OpenMP is main example here; OpenACC is similar
- **Syntax Approach**
  - Explicit Fork/Join
    - Cilk+, Go, CPlex, Chalice
  - Structured Parallelism – Safe, Implicit Fork/Join
    - Ada 2012/202X, ParaSail

What about Safety?

- **Safety can be achieved in two main ways:**
  - Add rules to make dangerous features safer -- *Ada, Rust*
  - Simplify language by removing dangerous features -- *SPARK, ParaSail*
- **Language-Provided Safety is to some extent orthogonal to approach to supporting parallel programming**
  - Harder to provide using Library Approach, but Rust does it by having more complex parameter modes and special semantics on assignment
  - Very dependent on amount of "aliasing" in the language
- **Level of safety determined by whether compiler:**
  - Treats programmer requests as orders to be followed -- *unsafe*
  - Treats programmer requests as checkable claims -- *safe*
- **If compiler can check claims, compiler can also *insert* safe parallelism automatically**

Library Approach – TBB, Java Fork/Join, Rust

- **Compiler is removed from the picture completely**
  - Except for Rust, where compiler enforces pointer ownership
- **Run-time library controls everything**
  - Focuses on the scheduling problem
  - May need some run-time notion of “tasklet ID” to know what work to do
- **Can be verbose and complex**
  - Feels almost like going back to assembly language
  - No real sense of abstraction from details of solution
  - Can use power of C++ templates to approximate syntax approach

TBB – Threading Building Blocks (Intel) using C++ templates

```
void SerialApplyFoo( float a[], size_t n )   //  Sequential Version
    { for( size_t i=0; i!=n; ++i ) Foo(a[i]); }

#include "tbb/tbb.h"
using namespace tbb;
class ApplyFoo {
   float *const my_a;
public:
   void operator()( const blocked_range<size_t>& r ) const {
      float *a = my_a;
      for( size_t i=r.begin(); i!=r.end(); ++i )
         Foo(a[i]);
   }
   ApplyFoo( float a[] ) : my_a(a) {}
};

#include "tbb/tbb.h”
void ParallelApplyFoo( float a[], size_t n ) {
   parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

Java Fork/Join

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class Sum extends RecursiveTask<Long> {
    static final int SEQUENTIAL_THRESHOLD = 5000;

    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }
```

Java Fork/Join (cont'd)

```
protected Long compute() {
      if(high - low <= SEQUENTIAL_THRESHOLD) {
          long sum = 0;
          for(int i=low; i < high; ++i)
              sum += array[i];
          return sum;
       } else {
          int mid = low + (high - low) / 2;
          Sum left  = new Sum(array, low, mid);
          Sum right = new Sum(array, mid, high);
          left.fork();
          long rightAns = right.compute();
          long leftAns  = left.join();
          return leftAns + rightAns;
       }
   }

   static long sumArray(int[] array) {
       return ForkJoinPool.commonPool().invoke(new Sum(array,0,array.length));
   }
}
```

The Rust Language

- **Rust is from Mozilla** *http://rust-lang.org*
  - From “browser” development group
    - Browser has become enormous, complex, multithreaded
  - C-ish syntax, but with more of a “functional” language feel
    - *Trait*-based inheritance and polymorphism; *match* instead of *switch*
  - Safe multithreading using *owned* and *managed* storage
    - *Owned* storage in global heap, but only one pointer at a time (no garbage collection)
      - Similar to C++ “Unique” pointers
      - Compile-time enforcement of ownership, including transfer
    - Originally also provided *Managed* storage in task-local heap, allowing many pointers within task to same object, but since dropped to avoid need for garbage collector
    - *Complex* rules about assignment and parameter passing
      - copy vs. move vs. reference semantics
      - temporarily borrowing (a ref) vs. permanently moving

The GNAT Pro Company

Example using Rust (and futures)

```
fn Word_Count
  (S : &str; Separators : &str)
  -> uint {
    let Len = S.len();
    match Len {
      0 => return 0; // Empty string
      1 => return if Separators.contains(S[0]) { 0 } else { 1 };
                     // One character string

      _ =>           // Multi-character string; divide and conquer
        let Half_Len = Len/2;
        let Left_Sum = future::spawn {
          || Word_Count(S.slice(0, Half_Len-1), Separators)};
        let Right_Sum = future::spawn {
          || Word_Count(S.slice(Half_Len, Len-1),
               Separators)};

        if Separators.contains(S[Half_Len]) ||
          Separators.contains(S[Half_Len+1]) { // Separator(s) at border
            return Left_Sum.get() + Right_Sum.get();    // read from futures
        } else {                               // Combine words at border
            return Left_Sum.get() + Right_Sum.get() – 1;// read from futures
        }
    }
 }
```

Simple cases

Divide and Conquer

Word Count Examples use “divide and conquer”

```
S: “This is a test, but| it’s a bit boring.”
            1111111111|2222222222333333333
   1234567890123456789|0123456789012345678
```

Separators: [‘ ’, ‘,’, ‘.’]

Word_Count(S, Separators) == ?

```
|S| == 38              // |…| means “length”
Half_Len == 19
Word_Count(S[1 .. 19], Separators) == 5
Word_Count(S[19 <.. 38], Separators) == 4
Sum == 9               // X <.. Y means (X, Y]
S[19] == ‘t’           // ‘t’ not in Separators
S[19+1] == ‘ ’         // ‘ ’ is in Separators
return 9
```

Pragma Approach – OpenMP, OpenACC

- **User provides hints vi #pragma omp/acc**
- **No building blocks – all smarts are in the compiler**
- **Not conducive to new ways of thinking about problem**
  - Language provides no new constructs, so no help in developing new kinds of solutions, new paradigms
- **Becomes a “pattern match” problem**
  - What patterns are recognized by compiler, and hence will be parallelized, vectorized, otherwise optimized

OpenMP example of summing array

```
#define N 10000 /*size of a*/
 void calculate(long *); /*The function that calculates the elements of a*/
 int i;
 long w;
 long a[N];
 calculate(a);
 long sum = 0;

 /*forks off the threads and starts the work-sharing construct*/
 #pragma omp parallel for private(w) reduction(+:sum) schedule(static,1)
 for(i = 0; i < N; i++)
   {
     w = i*i;
     sum = sum + w*a[i];
   }
 printf("\n %li",sum);
```

Two Main Syntax Approaches

- **Fork/Join Parallelism – Harder to enforce safety**
  - Go, Cilk+, CPlex, Chalice
- **Structured Parallelism – Implicit Fork/Join**
  - ParaSail
  - Ada 202x
- **Parallel “for” loops appear in both**

Why Add Syntax? To Escape the Sequential Mindset

- **Many problems are inherently parallel**
  - or at least not "fundamentally" sequential
- **Years of training in programming have taught us to think sequentially**
  - "a program is a sequence of instructions to the computer"
- **Need to "un-learn" these lessons**
  - try to preserve the natural parallelism of the problem in the solution
- **Use iterators to describe solution, without imposing an order**
  - this is why the "default" loop in ParaSail is unordered
  - must explicitly say "forward" or "reverse" if that is essential to solution
  - may want to separate out I/O of data from computation because of difficulty of doing "unordered" I/O

Biggest challenges to achieve “parallel” mindset

- **Avoiding global data**
  - And you are *not* allowed to just substitute one big shared data structure that is passed “everywhere” – (ParaSail compiler example)
  - Break global data up into pieces according to when it is created or updated (exclusive use), and when it is merely being read (shared use)
  - Use local data whenever possible by grouping creator and user
- **Separating (and trying to shrink) parts of program that *need* to operate sequentially from those that don’t**
  - Sequential output often needed to produce *deterministic*, *repeatable* result, hence …
  - *Sequentially* assign unique, ordered indices to each part of problem
  - Perform each part independently (in *parallel*), using unique index as appropriate
  - *Sequentially* emit final results in order of unique indices

Use high-level constructs

- Use sets, maps, iterators, comprehensions
- Keeps program out of the “housekeeping” and “single-element” activities which can create unnecessary sequential code

Explicit Fork/Join Syntax Approach -- Cilk, Go, CPlex, Chalice

- **Asynchronous function call**
  - cilk_spawn C(X) // Cilk
  - go G(B) // Go
  - _Task _Call F(A) // CPlex
  - fork tk := vr.call() // Chalice
- **Wait for spawned strand/goroutine/task**
  - cilk sync; or implicit at end of function
  - <implicit for Go>
  - _Task _Sync; or end of _Task _Block { … }
  - join tk;

Cilk+ from MIT and Intel

- **Keywords – Express task parallelism:**
  - *cilk_for* - Parallelize for loops
  - *cilk_spawn* - Specifies that a function can execute in parallel
  - *cilk_sync* – Waits for all spawned calls in a function
- **Reducers:**
  - Eliminate contention for shared variables among tasks by automatically creating views of them as needed and "reducing" them in a lock free manner
  - “tasklet local storage” + reduction monoid (operator + identity)
- **Array Notation:**
  - Data parallelism for arrays or sections of arrays.
- **SIMD-Enabled Functions:**
  - Define functions that can be vectorized when called from within an array notation expression or a #pragma simd loop.
- **#pragma simd: Specifies that a loop is to be vectorized**

Fibonacci example in Cilk+

```
int fib(int n)
{
    if (n < 2) {
        return n;
    }
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

Cilk+ quicksort example

```
void parallel_quicksort( T* first, T* last ) {
    while( last-first>QUICKSORT_CUTOFF ) {
        // Divide
        T* middle = divide(first,last);
        if( !middle ) return;

        // Now have two subproblems: [first..middle) and (middle..last)
        if( middle-first < last-(middle+1) )  {
            // Left problem [first..middle) is smaller, so spawn it.
            cilk_spawn parallel_quicksort( first, middle );
            // Solve right subproblem in next iteration.
            first = middle+1;
        } else {
            // Right problem (middle..last) is smaller, so spawn it.
            cilk_spawn parallel_quicksort( middle+1, last );
            // Solve left subproblem in next iteration.
            last = middle;
        }
    }
    // Base case
    std::sort(first,last);
}  // implicit cilk_sync
```

The Go Language

- **Go is from Google** *http://golang.org*
  - Rob Pike from early Bell Labs Unix design team
  - Quite "C" like syntactically but with some significant *differences*:
    - Object name *precedes* type name in syntax; allows type name to be omitted when can be inferred
      - e.g. "X int;" vs "int X;" ➔ "X := 3;" *// declares and inits X*
    - No pointer arithmetic; provides *array slicing* for divide-and-conquer
    - "Goroutines" provide easy *asynchronous function calls*; communicate via *channels* and *select* statements, but no race-condition checking built in
    - *Interfaces* and *method sets* but no classes
    - Fully garbage collected

Word Count Example using Go (with goroutines and channels)

Simple cases

Divide and Conquer

```
func Word_Count
  (s string; separators string) int = {
    slen := len(s)
    switch slen {
      case 0: return 0 // Empty string
      case 1:
        if strings.ContainsRune(separtors, S[0]) {
            return 0  // A single separator
        } else {
            return 1  // A single non-separator
        }
      default:     // Multi-character string; divide and conquer
        half_len := slen/2
        var left_sum = make(chan int)    // create a channel for left half
        var right_sum = make(chan int)   // create a channel for right half
        go func() {left_sum <- Word_Count(s[0:half_len], separators)}()
        go func() {right_sum <- Word_Count(s[half_len:slen], separators)}()

        if strings.ContainsRune(separators, rune(s[half_len-1])) ||
          strings.ContainsRune(separators, rune(s[half_len])) {
            // At least one separator at border
            return <-left_sum + <-right_sum     // read from channels
        } else {    // Combine words at border
            return <-left_sum + <-right_sum - 1 // read from channels
        }
    }
 }
```

Chalice from Microsoft Research (Rustan Leino)

- **Fork/join parallelism**
- **Permission "contracts" in pre- and postconditions**
  - requires **acc**(X) & **acc**(Z) & X > Z
  - requires **rd**(Y) & Y > 0
  - ensures **rd**(Y)   // gives access back
  - (may *dispose* of access, meaning object is destroyed)
- **"acc" is full r/w permission, "rd" is fractional r/o perm.**
- **Permissions also used for locking order**
  - requires **waitlevel** from.mu << to.mu
  - prevent deadlock by strict partial order of locks
- **Permissions carried in channels, and need credits for send and receive**
  - **where acc**(X) & **rd**(Y) // message carries r/w acc to X, r/o to Y
  - requires **credit**(chan, +2) // right to receive 2 messages
  - requires **credit**(chan, -3)   // obligation to send 3 messages

Structured Parallelism Approach – Implicit Safe Fork/Join

```
 parallel                           -- Ada 202X
     sequence_of_statements
{and
     sequence_of_statements}
 end parallel;
```

or…

```
     sequence_of_statements
{||                                   // ParaSail
     sequence_of_statements}
```

Word_Count example using “heavy weight” tasks:

```
function Word_Count(S : String; Separators : String) return Natural is
    use Ada.Strings.Maps;
    Seps : constant Character_Set := To_Set(Separators);

    task type TT(First, Last : Natural; Count : access Natural);
    subtype WC_TT is TT;  --  So is visible inside TT
    task body TT is begin
       if First > Last then     --  Empty string
          Count.all := 0;
       elsif First = Last then  --  A single character
          if Is_In(S(First), Seps) then
             Count.all := 0;    --  A single separator
          else
             Count.all := 1;    --  A single non-separator
          end if;
       else  --  Divide and conquer
          ... See next slide
       end if;
    end TT;

    Result : aliased Natural := 0;
 begin
    declare  --  Spawn task to do the computation
       Tsk : TT(S'First, S'Last, Result'Access);
    begin
       null;
    end;  --  Wait for subtask
    return Result;
 end Word_Count;
```

Simple cases

Start outer task

“Heavy” Word_Count example (cont’d):

```
 function Word_Count(S : String; Separators : String) return Natural is
     use Ada.Strings.Maps;
     Seps : constant Character_Set := To_Set(Separators);
     task type TT(First, Last : Natural; Count : access Natural);
     subtype WC_TT is TT;  --  So is visible inside TT
     task body TT is begin
        if …  --  Simple cases (see previous slide)
        else  --  Divide and conquer
           declare
              Midpoint : constant Positive := (First + Last) / 2;
              Left_Count, Right_Count : aliased Natural := 0;
           begin
              declare  --  Spawn two subtasks for distinct slices
                 Left : WC_TT(First, Midpoint, Left_Count'Access);
                 Right : WC_TT(Midpoint + 1, Last, Right_Count'Access);
              begin
                 null;
              end;  --  Wait for subtasks to complete

              if Is_In(S(Midpoint), Seps) or else
                Is_In(S(Midpoint+1), Seps) then  --  At least one separator at border
                 Count.all := Left_Count + Right_Count;
              else  --  Combine words at border
                 Count.all := Left_Count + Right_Count - 1;
              end if;
           end;
        end if;
     end TT;
     … See previous slide
end Word_Count;
```

Divide and Conquer

Word_Count example using structured light-weight construct:

```
function Word_Count (S : String; Separators : String) return Natural
  with Global => null, Potentially_Blocking => False is
    case S'Length is
     when 0 => return 0; --  Empty string
     when 1 =>           --  A single character
       if Is_In(S(S'First), Seps) then
          return 0;      --  A single separator
       else
          return 1;      --  A single non-separator
       end if;
     when others =>
       declare           --  Divide and conquer
          Midpoint : constant Positive := (S'First + S'Last) / 2;
          Left_Count, Right_Count : Natural;
       begin
          parallel       --  Spawn two tasklets for distinct slices
             Left_Count  := Word_Count (S(S'First .. Midpoint), Separators);
          and
             Right_Count := Word_Count (S(Midpoint+1 .. S'Last), Separators);
          end parallel;  --  Wait for tasklets to complete

          if Is_In(S(Midpoint), Seps) or else
            Is_In(S(Midpoint+1), Seps) then  --  At least one separator at border
             return Left_Count + Right_Count;
          else                               --  Combine words at border
             return Left_Count + Right_Count - 1;
          end if;
       end;
    end case;
end Word_Count;
```

Simple cases

Divide and Conquer

Parallel Loop

```
for I in parallel 1 .. 1_000 loop
   A(I) := B(I) + C(I);
end loop;

for Elem of parallel Arr loop
   Elem := Elem * 2;
end loop;
```

Parallel loop is equivalent to parallel block by unrolling loop, with each iteration as a separate alternative of parallel block.

Compiler will complain if iterations are not independent or might block (again, using Global/Nonblocking aspects)

Parallel Loop Issues

- **Exiting the block/loop, or a return statement**
  - All other tasklets are aborted (need not be preemptive) and awaited, and then, in the case of return with an expression, the expression is evaluated, and finally the exit/return takes place.
  - With multiple concurrent exits/returns, one is chosen arbitrarily, and others are aborted.
- **Handling arrays with many elements with small amount of work to be done on each element**
  - Compiler may choose to “chunk” the loop into subloops, each subloop becomes a tasklet (subloop runs sequentially within tasklet).
- **Accumulating results without excessive synchronization on accumulators**
  - Special support for map/reduce

Safety through Simplification – SPARK 202X and ParaSail

- **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

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

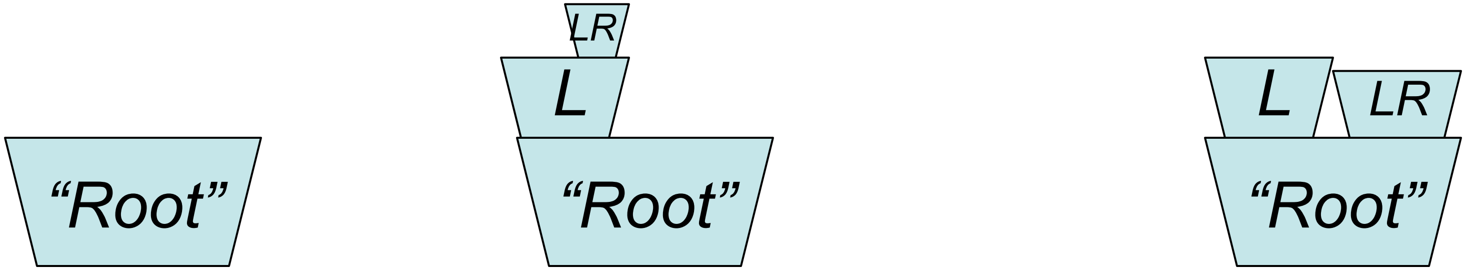
Expandable Objects Instead of Pointers to Avoid Aliasing

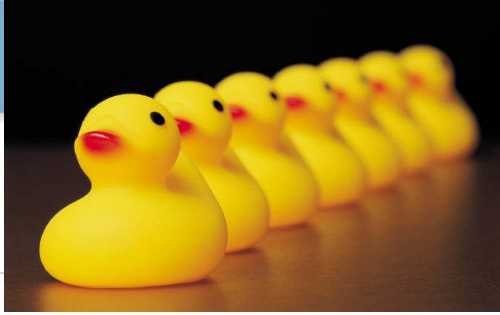
- **All types have additional null value; objects can be declared optional (i.e.null is OK) and can grow and shrink**
  - Eliminates many of the common uses for pointers, e.g. trees
  - Assignment (“:=“) is by copy
    - Move (“<==“) and swap (“<=>”) operators also provided
- **Generalized indexing into 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
- **SPARK 202X and Rust use pointer ownership instead**
  - More complex semantic model than expandable objects
  - But more familiar to Ada and C programmers

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
```





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.

Syntax for parallel Map/Reduce in ParaSail (and Ada 202X?)

- 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))
```

Review of Syntax Approach to Parallelism

- **Cilk+, Go, CPlex, Chalice**
  - Use asynchronous function call + synchronize (explicit or implicit)
  - Few safety checks in Cilk+/Go/Cplex; full checks in Chalice
  - Provides safe communication mechanisms such as Go’s channel and Cilk’s Reducer
  - Cilk’s Reducer provides “tasklet local storage” so each “chunk” gets its own accumulator
  - Reducer uses “monoid” = operator + identity (aka “zero”)
- **Ada 202X**
  - parallel block: “parallel F(X); and G(Y); end parallel”
  - parallel loop with array of partial results with one element per chunk
- **ParaSail – safety through simplification**
  - No globals, no pointers, no parameter aliasing, expandable objects
  - Implicit safe parallel semantics for all operators
- **Both Ada 202X and ParaSail:**
  - Syntactic Sugar for many kinds of iterators

Enforcing Safety in a Parallel Program – Data Races

- **Data races**
  - Two simultaneous computations reference same object and at least one is writing to the object
  - Reader may see a partially updated object
  - If two Writers running simultaneously, then result may be a meaningless mixture of two computations
- **Solutions to data races**
  - Dynamic run-time synchronization to prevent simultaneous use
    - Use full locks or atomic hardware instructions such as compare-and-swap
  - Static compile-time checks to prevent simultaneous incompatible references – depends on constraints on aliasing
- **Can support both**
  - Dyamic: Chalice “monitor” objects; ParaSail “concurrent” objects
  - Static: Chalice permissions in preconditions; Rust “borrowing”; ParaSail hand-off semantics plus no globals; SPARK anti-aliasing checks
  - Reminiscent of capability-based systems: compile-time or run-time

Safely solving the data race problem

Data-Race safety mantra from *Niko Matsakis* of Rust fame:

=> Aliasing, Mutability, Concurrency – *pick any two*

- **No Aliasing**
  - Occam, Erlang – "shared nothing" – distributed programming
- **Immutable Data**
  - Haskell - Pure functional language
- **Mostly Immutable Data + All shared data synchronized**
  - Clojure – Mostly functional, optimistic transaction-based sync on shared data
- **No user-visible parallelism**
  - APL, Matlab – parallelism *inside* vector/matrix operations of language
- **Constrained Aliasing**
  - SPARK, Rust, ParaSail, Chalice, Ada 202X
  - Can use both Compile-Time and Run-Time checks

Constrained Aliasing

- **SPARK**
  - No pointers – array indexing for bounded “linked” data structures
  - Global annotations establish the frame
  - No aliasing on parameter passing
- **Rust**
  - Pointer ownership transferred by assignment A = B // B now dead
  - Reference as way to temporarily grant access X = &mut Y
    - Borrowing checker to make sure exclusive mut references
- **ParaSail**
  - No global variables and no pointers
  - Handoff semantics on parameter passing based on “mode”
- **Chalice**
  - acc(X) & rd(Y) in preconditions and postconditions
  - permissions at activation-record level – no finer granularity of ownership
- **Ada 202X**
  - Global annotations
  - Anti-aliasing preconditions using X’Overlaps_Storage(Y)

Global annotations in Ada 202X

```
Global => in out all -- default for non-pure pkgs
Global => null         -- default for pure packages

-- Explicitly identified globals with modes
Global => (in P1.A, P2.B,
           in out P1.C,
           out P1.D, P2.E)

-- Pkg data, access collection, task/protected/atomic
Global => in out P3             -- pkg P3 data
Global => in out P1.Acc_Type    -- acc type
Global => in out synchronized -- prot/atomic
```

Can add run-time guarded objects for flexibility

- **Monitors in Chalice**
  - Three states for objects: *non-monitor, available, held*
- **Concurrent objects in ParaSail**
  - external view – aliasing permitted, but no access
  - internal view – gained implicitly when calling an operation
    - back to compile-time checks when inside the operation
- **Protected/Atomic objects in SPARK**
  - external view – aliasing permitted, but no access
  - internal view – no parallelism

Safety in a Parallel Program -- Deadlock

- **Deadlock, also called "Deadly Embrace"**
  - One thread attempts to lock A and then lock B
  - Second thread attempts to lock B and then lock A
- **Solutions amenable to language-based approaches**
  - Chalice: Assign full order to all locks; must acquire locks according to this "waitlevel" order
  - ParaSail/SPARK: Localize locking into "monitor"-like construct and check for cyclic locking
- **More general kind of deadlock – waiting forever**
  - One thread waits for an event to occur, but event never occurs
  - Chalice solution based on send/receive "credit"
  - ParaSail/Ada 202X: Identify where blocking might occur
    - Nonblocking aspect in Ada 202X
    - **queued** qualifier in ParaSail

Nonblocking aspect in Ada 202X; “queued” in ParaSail

- **Ada 202X Nonblocking aspect**

```
procedure Suspend_Until_True
  (S : in out Suspension_Object)
     with Nonblocking => False;

package Ada.Characters.Handling
     with Nonblocking => True is …
```

- **ParaSail “queued” qualifier**
  - Note that default is non-blocking

```
queued func Delay_Until(Until : Time);
```

Conclusions on Approaches to Safe Parallel Programming

- **Library, Pragma, and Syntax Approaches Possible**
- **Safety can be provided in any approach**
- **Safety can come from more rules, or fewer features**
- **New kinds of contracts may be needed**
  - Global annotations (if globals are allowed at all)
  - Pointer ownership contracts (Rust, SPARK 202X?)
  - Extended parameter modes/permissions (Chalice, ParaSail)
  - Non-Blocking contracts (Ada 202X, ParaSail)
  - Send/Receive Credits (Chalice)
- **Language support for parallel programming should**
  - try to help programmers "escape" the sequential mindset
  - help think in terms of overall problem to be solved, not the order to solve it
  - identify all possible data races and deadlocks at compile time to keep the problem tractable