



The Sisal Model of Functional Programming and its Implementation

Jean-Luc Gaudiot, et al.

Presented by Nick Rutar



Sisal FAQ

- What does Sisal stand for?
 - Stream and Iteration in a Single Assignment Language
- What kind of language is it?
 - Sisal is a functional programming language
- Where did it come from?
 - Developed by
 - Lawrence Livermore National Laboratory
 - Colorado State University
 - University of Manchester
 - Digital Equipment Corporation
- Why are we talking about it in this class?
 - Sisal was developed with a goal to “design a general-purpose implicitly parallel language for a wide range of parallel platforms”

Sisal overview

- User-defined names are “identifiers”, not variables
 - Refer to values, not memory locations
- All values produced and used are dynamic
 - Identifiers bound only for duration of execution
- All expressions return values based on
 - Values bound to formal arguments
 - Constituent identifiers
- No side effects
- Allows richer analyses of program than for imperative languages

Sample Code

```
type OneDim = array [ real ];
type TwoDim = array [ oneDim ];
function generate( n: integer
                 returns TwoDim, TwoDim )
  for i in 1, n cross j in 1, n
  t1 := real(i) * real(j);
  t2 := real(i) / real(j);
  returns array of t1
         array of t2
  end for
end function % generate
```

Parallelism in sample code

- All Sisal expressions evaluate to value sets
 - Function evaluates to two arrays
- for expression indicates potential parallelism to the compiler
 - Body of loop instantiated as many times as values in index range
 - Each body instantiation is independent
 - No data dependencies
- Independent loop bodies may be done in parallel

OSC: Optimizing Sisal Compiler

- Compilation proceeds through various stages
 - Translates Sisal source to data flow graph language, IF1
 - Translate IF1 to annotated memory management graph language, IF2
 - C code is produced from IF2
 - Target machine compiler invoked
- Various options for compilation & execution
 - Optimization
 - Output (Syntax error messages)
- Available for most architectures

Sisal compiler issues

- Update in place and copy elimination
- Build in Place
- Reference Counting Optimization
- Vectorization
- Loop Fusion
- Double Buffering Pointer Swap
- Inversion

Update in place/copy elimination

```
let
  A: = array[1: 1,2,3];
  B: = A[2 : 999];
in
  A, B
end let
```



Alternate replacement

```
C := array[1: 1,2,3,4,5]
T0 := C[3]; T1 := C[4];
D := C[3: T1];
E := D[4: T0];
```

Return [1:1,2,3],[1:1,999,3]

Use containers instead of
throwing them away

Problem: Multiple copies of the array
Swapping two elements even worse

Build in Place

L := F(0,A[1],A[2]);

R := F(A[N-1],A[N],0);

III:= for i in 2, n-1 ...

LIII = array_addl(III,L);

LIIR = array_addh(LIII,R);

Create a Buffer

.....
L....
L...R
LIIR



Requires allocation and unneeded data movement

Reference Counting & Vectorization

■ Reference Counting

- Reference counts show when
 - A value can be updated in place
 - Value's memory can be recycled
- Can be expensive, especially on parallel machines
- OSC eliminates most reference counting
 - Lifetime analysis
 - Operation merging

■ Vectorization

- Sisal's underlying dataflow representation make loops easy to move
 - This means OSC can vectorize extremely well

Loop Fusion

T0 := for i in 1, n returns
array of A[i]*2
end for

T1 := for i in 1, n returns
array of B[i]*3
end for

X := for i in 1, n returns
array of T0[i] + T1[i]
end for

X := for i in 1, n returns
array of A[i]*2+B[i]*3
end for



Double Buffering Pointer Swap

for initial

A := start_values()

while not done(A) repeat

A := time_step(old A)

Allocates new buffer each
time step

Allocate initial buffer
outside loop and pointer
swaps initial and secondary
buffers



Inversion

```
X:= for i in 1,n
  v:= if i = 1 then %Left
      elseif i = n then %Right
      else %inner
  end if
X0:= for i in 1,max(0,n) %left
X1:= for i in 1,max(0,n) %right
X2:= for i in 2,n-1 %inner
X:= X0 || X2 || X1
```

If-tests introduce large overhead and inhibits parallelism

D-OSC

- Extension of OSC for distributed-memory machines
- Code generation produces C plus MPI calls
- Master process divides parallel loops into slices
 - Slices executed in parallel by designated processes
- D-OSC implemented in four phases

D-OSC Phases

- Phase 1: Base
 - No analysis, naïve code generation
 - Arrays and loops distributed among processors
 - Unique array identifiers explicitly created with table on each processor
- Phase 2: Rectangular Arrays
 - Higher dimensional arrays of arrays replaced by rectangular arrays
- Phase 3: Block Messages
 - Algorithm to obtain block messages instead of individual array elements
- Phase 4: Multiple Alignment
 - Reduces number of messages by creating overlapping array sections

D-OSC Results

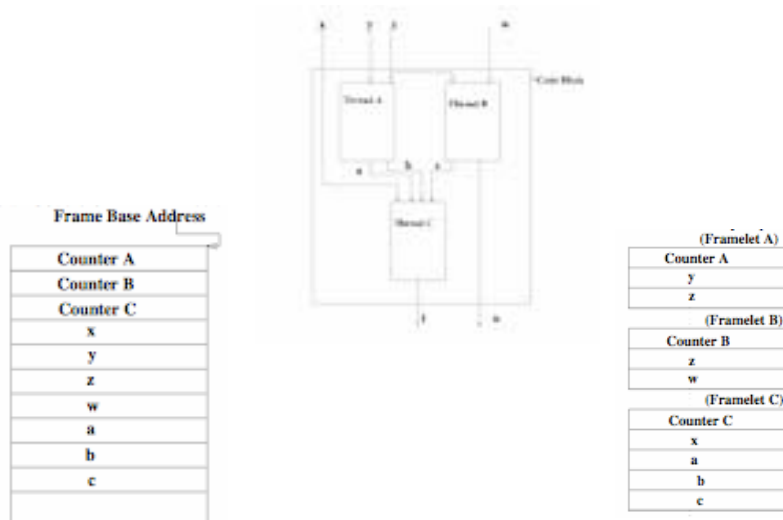
- Livermore loops on network of 4 workstations
- 1st number - messages passed, 2nd num - volume

Program	Type	Base	RecArrays	BlkMsgs	Multi
ii1	1D	6605, 132132	6603, 211368	603, 31368	303, 12168
ii2	1D	6443, 126656	6443, 213128	6443, 213128	6443, 213128
ii3	1D	3, 96	3, 168	3, 168	3, 168
ii6	1D, 2D	10833, 213036	13223, 430408	13223, 430408	13223, 430408
ii7	1D	4807, 86568	7503, 225168	953, 18968	303, 8568
ii9	2D	5883, 117136	2403, 76968	603, 28968	603, 28968
ii12	1D	9005, 180132	3003, 96168	1503, 24168	3, 168
ii21	2D	471, 8520	14403, 460968	123, 58728	123, 58728
ii24	1D	29703, 594096	29703, 950568	29703, 950568	29703, 950568

Multithreaded Execution

- Sisal suited as source for multithreaded code
- Two models considered
 - Blocking Thread Model
 - Thread may be suspended and resumed later
 - Architecture must support context switching
 - Rely on Frame model
 - Storage statement associated with each invocation of a code-block
 - Non-blocking Thread Model
 - Thread starts and runs until termination
 - Relies on Framelet model
 - Fixed size unit of storage associated with each thread instance
 - Data values shared between threads are replicated

Blocking & Non-Blocking Example



Multithreaded Code Generation

- **Converts programs into two intermediate forms**
 - MIDC-2 (non-blocking)
 - MIDC-3 (blocking)
 - Both derived from MIDC (Machine Independent Dataflow Code)
- **Guided by**
 - Minimize synchronization overhead
 - Maximize intra-thread locality
 - Assure deadlock-free threads
 - Preserve functional and loop parallelism in programs

Multithread Code Gen Phases

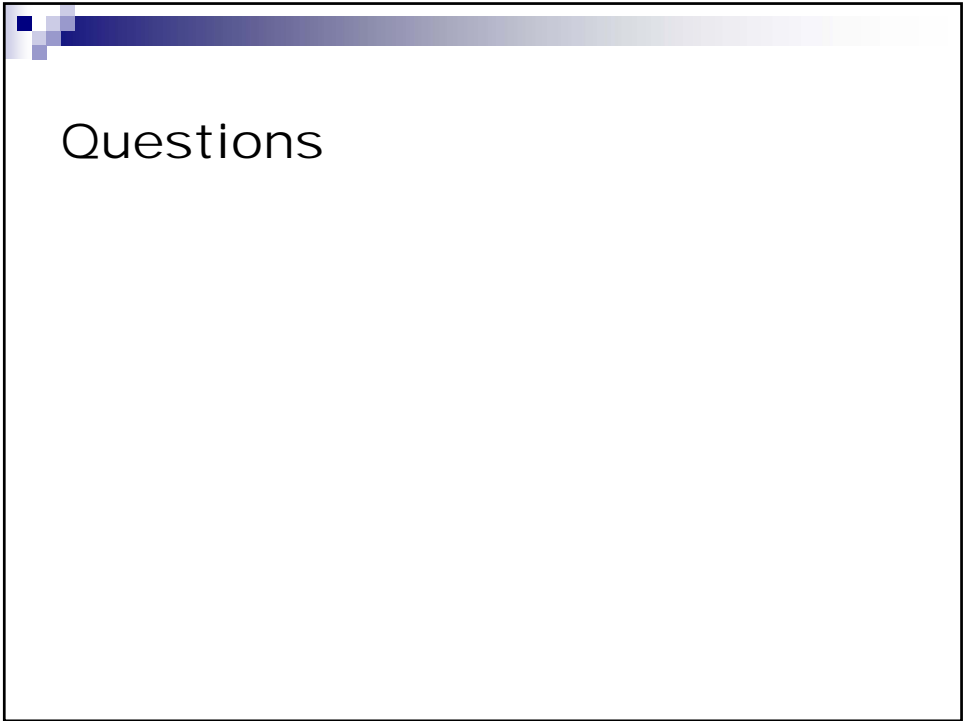
- **Phase 1**
 - Same for blocking & non-blocking
 - Compile Sisal program to IF2 using OSC
- **Phase 2**
 - Handle remote memory accesses as split-phase or single-phase



Effect of Network & Memory



Conclusions



Questions