Conf-9504126--1

UCRL-JC-119529
PREPRINT

Implementation and Performance of the Pseudoknot Problem in Sisal[1]

John T. Feo
Lawrence Livermore National Laboratory

and

Melody Ivory
University of California, Berkeley

This paper was prepared for submittal to the
Conference on High Performance
Functional Computing,
Denver, CO
April 9-11, 1995

December 1994

TJ

Implementation and Performance of the Pseudoknot Problem in Sisal[1]

John Feo
Computer Research Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

Melody Ivory
Department of Computer Science
University of California, Berkeley
Berkeley, CA 94720

***Abstract**: The Pseudoknot Problem is an application from molecular biology that computes all possible three-dimensional structures of one section of a nucleic acid molecule. The problem spans two important application domains: it includes a deterministic, backtracking search algorithm and floating-point intensive computations. Recently, the application has been used to compare and to contrast functional languages. In this paper, we describe a sequential and parallel implementation of the problem in Sisal. We present a method for writing recursive, floating-point intensive applications in Sisal that preserves performance and parallelism. We discuss compiler optimizations, runtime execution, and performance on several multiprocessor systems.*

1. Introduction

The Pseudoknot Problem [2] is a modification of a three-dimensional molecular biology application. The program exhaustively searches a state space of structures, built from nucleotides stored in a database, to build all possible three-dimensional structures of one section of a nucleic acid molecule. Only structures that satisfy certain constraints are returned. Both the database and the structural constraints are built into the program. Recently, the

[1] This work was supported, in part, by Lawrence Livermore National Laboratory and the Department of Energy Technology Transfer Initiative under DOE contract W-7405-ENG-48. This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

application has been used to compare and to contrast functional languages [3]. The problem is appropriate for language comparison for several reasons:

1. The problem spans two important application domains; it includes a deterministic, backtracking search algorithm and floating-point intensive computations. The search path extends 23 steps and includes five branch points. The C version of the program executes 6.9 million floating point operations and 190 thousand square root and trigonometric functions.

2. The width of the search tree is quite large for even small problems. A particular expression of the problem may expose too much parallelism degrading performance.

3. The program includes both stack and array data structures. In a parallel environment, the traditional operations on these structures must be defined without introducing race conditions and excessive copying.

4. The computation includes opportunities for lazy and eager evaluation. The program may compute a nucleotide's position in a structure when it is added to the structure, or it may compute the position only when needed.

In this paper, we present a sequential and a parallel version of the Pseudoknot Problem written in Sisal [4], a high-performance functional language developed by Lawrence Livermore National Laboratory and Colorado State University. The performance of the sequential version was first reported in [3]. The objective of the Sisal Language Project is to develop high-performance functional compilers and runtime systems for commercially available computer systems. The language developers have focused on large-scale scientific application codes. Currently, mature Sisal compilers exist for single processor and shared-memory multiprocessor systems, including the family of Cray computers, Sun and SGI systems, and IBM and Macintosh PCs. On such systems, Sisal codes can execute as fast as codes written in imperative programming languages [1].

We are experienced at writing Sisal programs with many more floating-point operations than necessary to solve the Pseudoknot Problem; however, we have little experience writing recursive and/or stack-base programs. This paper is the first substantive report on the expression, optimization, and execution performance of a recursive, stack-based Sisal program. In section two we present a sequential and a parallel implementation of the pseudoknot problem written in Sisal. While we can compile the latter for sequential execution, we wrote two versions of the program to minimize the sequential execution time. Section three discusses compilation and runtime issues and gives the execution times of the programs on several multiprocessor computer systems. In section four, we summarize and present our conclusions.

2. Sisal implementation

The pseudoknot program returns all possible three-dimensional structures of one section of a nucleic acid molecule by adjoining nucleotides one at a time Since the order in which we can add nucleotides to the structure is known, the search is determinant. Figure 1 depicts a partial search tree and gives the names of the Sisal routine called at each level. The parameter indicates the type of nucleotide, A, C, G, or U, considered at that level. Before adding a nucleotide to the structure, the program decides whether the addition violates the constraints of the problem. If the addition is legal, the program adds the nucleotide to the structure and descends to the next level; otherwise, the program considers an alternative nucleotide at the same level. If there are no other alternatives, the last nucleotide added to the structure is removed and the program ascends to the previous level. When the search reaches level 23, a solution has been found by definition. The program saves the structure, removes the last nucleotide, and pops back one level. Note that multiple alternatives exist only at steps 16, 17, 18, 20, 21, and 22. These steps represent the branch points in the search space. Pruning occurs only at steps 18 and 22, where the addition of a nucleotide may violate the problem's constraints.

2.1 Sequential implementation

While we could have implemented the sequential program as a simple sequence of instructions, we decided to write a recursive, depth-first search program. This implementation is similar to the original C program and the programs written in other functional languages. The recursive program was not difficult to write in Sisal; however, we did have to think carefully about the data structures we used and memory management issues.

There are three important data structures in the program: 1) the database of nucleotides, 2) the stack maintaining the current structure, and 3) the list of solutions. The original C program implements the database as an array of nucleotides. It defines a nucleotide as a variant structure to accommodate the differences between the four types without introducing multiple data types. Although Sisal supports the equivalent of variant records, we chose to represent a nucleotide as a real, two-dimensional array. The difference between the four nucleic types manifests itself in the number of columns in the fifth row. Since array type definitions in Sisal are independent of size and shape, we can define a single data type for all four nucleic types. It is important that all nucleotides have the same data type because Sisal functions are not polymorphic. The program calls the same function at different sites with different nucleic types. For example, the program calls **wc_dumas** with a nucleotide of type U at level 1, but with nucleotide of type C at level 3.

As the program descends and ascends the search tree, nucleotides are added to and removed from the stack which maintains the current structure. We implemented the stack as an array, and used the array routines **array_addh** and **array_remh** to push and pop nucleotides. Since Sisal implements n-dimensional arrays as an array of pointers to (n-1)-dimensional arrays, the stack is actually an array of pointers to the nucleotides in the database. No nucleotide is every copied; only addresses are copied. Moreover, Sisal passes all arrays by reference, so the stack is not copied at each call site. Copying can occur only when the stack is pushed or popped, and then only if the stack's reference count (number of outstanding consumers) is greater than one. If we can limit the number of consumers that modify the stack to one in every scope, the program will not copy.

Consider the Sisal code[2] for functions **pseudoknot_domains** and **wc_dumas**,

```
function pseudoknot_domains(k, stack, ... returns solutions_type)
  if k = 0 then
     reference(k, nucleotide, stack, ..., solutions)
  elseif k = 1 then
     wc_dumas (k, nucleotide, stack, ..., solutions)
  elseif k = 2 then
     helix3   (k, nucleotide, stack, ..., solutions)
  . . .
  . . .
  end if
end function
```

[2] To improve readability, we present only pseudocode, and not actual Sisal code.

```
function wc_dumas(k, nucleotide, stack, ..., solutions
          returns solutions_type)
   if pseudoknot_constraint(nucleotide, stack, ...) then
      let new_stack := array_addh(stack, nucleotide)
      in  pseudoknot_domains(k+1, new_stack, ..., solutions)
      end let
   else
      solutions
   end if
end function % wc_dumas
```

Pseudoknot_domains is the program's central function. It is a conditional statement that calls the function to be executed at each stage of the search. It is called originally from the main program with $k = 1$ and an empty stack. The functions **reference, wc_dumas, G37_A38, helix3,** and **helix5** are all similar; **P_O3** is different and is discussed later. Notice the recursive call to **pseudoknot_domains** in the *let* statement in **wc_dumas.**

In **pseudoknot_domains**, for a given value of **k**, there is only one consumer of **stack.** In **wc_dumas** there are two consumers of **stack**, but only the second modifies the stack and it is guaranteed to execute after the first. Thus, the **array_addh** operation executes in place without copying. Moreover, the recursive call to **pseudoknot_domains** is the only consumer of **new_stack** and so, it is passed with a reference count of one.

It might appear that the stack must be copied at the branch points of the search, but this is not so. Consider the following sequential implementation of **P_O3**

```
function P_O3(k, set_of_nucleotide, stack, solutions_in, ...
     returns solutions_type)
  for initial
    i          := 0;
    solutions  := solutions_in
  while i < array_size(set_of_nucleotide) repeat
    i          := old i + 1;
    nucleotide := set_of_nucleotide[i];
    solutions  := P_O3a(k, nucleotide, stack, ..., old solutions)
  returns value of solutions
  end for
end function % P_O3
```

where the code for **P_O3a** is similar to the code for **wc_dumas** and the other functions. Since each instance of the loop body is a consumer of **stack**, the object is not mutable by

P_O3a; consequently, the **array_addh** operation within **P_O3a** will copy the stack. We can eliminate the copy by explicitly passing the stack from iteration to iteration. This change requires that **P_O3a** returns both a stack and a solution list, and that it pop off the nucleotide that it pushes onto the stack before returning. With these changes, the code for **P_O3** and **P_O3a** is now

```
function P_O3(k, set_of_nucleotide, stack_in, solutions_in, ...
      returns stack_type, solutions_type)
  for initial
    i         := 0;
    stack     := stack_in;
    solutions := solutions_in
  while i < array_size(set_of_nucleotide) repeat
    i          := old i + 1;
    nucleotide := set_of_nucleotide[i];
    stack, solutions :=
       P_O3a(k, nucleotide, old stack, ..., old solutions)
  returns value of stack
          value of solutions
  end for
end function % P_O3

function P_O3a(k, nucleotide, stack, solutions, ...
          returns stack_type, solutions_type)
   if pseudoknot_constraint(nucleotide, stack, ...) then
      let
         stack1    := array_addh(stack, nucleotide);
         stack2,
         solutions := pseudoknot_domains(k+1, stack1, ..., solutions)
      in
         array_remh(stack2), solutions
      end let
   else
      stack, solutions
   end if
end function % P_O3a
```

Of course, we must make similar changes to **wc_dumas** and the other functions to maintain program consistency.

By explicitly removing the nucleotide pushed onto the stack before the call to **pseudoknot_domains**, **P_O3a** returns the same stack that it was passed. More importantly, there is now a single consumer of the stack in every scope. The compiler recognizes this fact and generates code to update the stack in place. The Sisal program executes with a single stack—all pushes and pops executing in place. Note that the Sisal code is no more complex than the equivalent imperative code in which the programmer also explicitly pushes and pops nucleotides onto and off the stack.

The third data structure of importance is the solution list. It is an array of solutions, or stacks. At level 23, the runtime system copies the stack and appends a pointer to the copy to the solution list. Remember that the stack is an array of pointers, so only memory addresses are copied. Note that the copy is unavoidable and language independent—the stack must be saved at this point. As with the stack, we single-threaded the solution list to eliminate any unnecessary copying; however, some copying might still occur. Since the number of solutions is not known, neither the compiler nor the runtime system can preallocate storage for the solution list. When the size of an array is unknown prior to definition, the Sisal runtime system preallocates n bytes of storage for the array (a runtime parameter with a default value of 100). As array elements are computed, they are written to the storage. If the storage is exhausted before the array is defined fully, the runtime system preallocates a greater amount of storage, copies the array elements already defined from the old storage to the new storage, recycles the old storage, and continues.

The final memory management issue regards scalarization. The program computes a large number of atomic positions: x, y, and z coordinates. Storing the coordinates as a record or an array results in the allocation and deallocation of a large number of small data objects. To avoid this cost, the program passes all coordinates as scalar values.

2.2 Parallel implementation

Parallel work occurs at the branch points of the search space. We exploit this parallelism by substituting a *for* expression in place of the *for initial* expression in **P_O3**,

```
function P_03_L(k, set_of_nucleotide, stack_in, solutions_in, ...
      returns stack_type, solutions_type)
  let
     sltns := for nucleotide in set_of_nucleotide
                sol_0        := array solutions_type [];
                stack, sltn := P_03a(k, nucleotide, stack_in, sol_0)
              returns array of sltn
              end for
  in
      stack_in,
      for sltn in sltns returns value of catenate sltn end for
  end let
end function % P_03_L
```

Each search branch rooted by a nucleotide in the set will execute concurrently. The system will give each thread its own copy of the stack; again only memory addresses are copied. This copying is unavoidable since each thread must be able to modify its stack independent of the other threads. Each sequential thread manipulates its stack in place without copying.

Given the large number of possible search paths, we must be careful not to create too many parallel threads. Ideally, we want to use **P_O3_L** at each call site and have the system decide whether or not to search the alternate paths sequentially or in parallel. Unfortunately, the current Sisal runtime system does not throttle parallelism. The decision as to how to execute the loop bodies of a *for* expression is made at compile time and depends on the loop's nesting level, the number of loop bodies, the size of the body, and the granularity of the target machine. On our target machines and for the given instance of the problem, we found that calling **P_O3_L** at level 16 and **P_O3** at levels 17, 18, and 20 gave the best performance. The static encoding of parallelism in a Sisal program is unsatisfactory; future Sisal runtime systems should automatically throttle parallel work permitting programmers to write more general code.

3. Performance

Here we report on the performance of the Sisal codes on a four processor SGI IRIS 340 and a sixteen processor Cray C-90. In the final version of the paper, we will also include a twelve processor SGI Challenge and possibly other machines that may become available to us in the coming months. For each test, we give the compiler and runtime parameters, the compile time, the execution time, and the data space used. The compile time includes both user and system time. The execution times of the programs are reported in *knots*. The unit gives the relative speed with respect to the execution speed of the sequential C program. To run at "100 knots" means to run at exactly the same speed as the sequential C program. All runs were double-precision. We used the Sisal compiler OSC 13.0 in all cases.

The compiler options for the sequential and parallel programs in all cases were, respectively,

```
    -cpp -seq -O -externC atan2
and
    -cpp -O -externC atan2.
```

The first option calls the C preprocessor before calling the Sisal frontend. The option -seq compiles the Sisal code for sequential execution. Code to spawn parallel work and manage shared resources among worker processes is not inserted in the C code generated by the Sisal compiler. -O turns on all Sisal optimizations. The -externC flag provides a convenient mechanism to link in C library routines. A similar flag is available for Fortran library routines. These flags are one component of the Sisal Foreign Language Interface.

For sequential runs, the only runtime flag we used was -r. This flag instructs the Sisal runtime system to generate a statistics file. We report herein the execution times and data space sizes from that report. For parallel runs, we used the -r, -w, and -ls flags. The second and third flags define the number of workers and the number of loop slices.

Table 1 gives the performance for the sequential and parallel Sisal code on a four processor SGI IRIS 340. The processors are 33MHZ MIPS 3000 chips with 64KB caches. The machine has a total of 64MB of main memory. The SGI runs under the IRIX 4.0.5 System V operating system. We used the C compiler gcc 2.5.8, and compiled all C programs at optimization level -O. The pseudoknots achieved by the Sisal program are respectable, but the worst that we have seen. The small cache size of the SGI machine may be hurting the Sisal performance. We saw similar performance with respect to the C code on a Sun4 with a 64K cache [3]. The C code generated by the Sisal program is voluminous causing Sisal programs to have worse instruction cache performance than equivalent C programs [5].

We suspect that the parallel code performs significantly worse on one processor than the sequential version because of copying, but we have not confirmed this suspicion. Performance on more than two processors was improved by slicing the *for* expression in **P_O3_L** in ten slices (one slice per search path at level 16). This number of slices resulted in a better load balance, and only slightly increased the runtime overhead. Even so, the speedup is below average for the SGI machine. Notice the slight increase in space per additional worker.

Table 2 gives the performance for the sequential and parallel Sisal code on a sixteen processor Cray C-90. This system is a vector supercomputer with 16GFlops peak performance. Long vector computations run fast on this machine; everything else runs significantly below peak performance. The system has 128MW of main memory arranged in 64 banks per processor, and no cache. The operating system is UNICOS 7.C. We used the C compiler cc 4.0.2.8, and compiled all C programs at optimization level -O2. The parallel program runs slightly worse on one processor than the sequential program. We believe the degradation is small because the Sisal compiler splits the *for* expression in **P_O3_L** into two loops: a concurrent–vector loop and a concurrent loop. The former computes the coordinates of the nucleotides appended to the structure at step 16, and passes the coordinates in an array to the second loop. The second loop then descends the alternate search paths in parallel. Since we use a *for initial* expression at step 16 in the sequential version, this optimization is not available. Increasing the amount of vector work in the parallel code compensates for any loss in performance due to parallel runtime overhead.

Given the small size of the computation and small vector lengths, the Sisal program exhibits good performance and speedup. The data space is larger than on the SGI, but still small. Note that on the Cray, all words are 8 bytes long. Neither increasing the number of parallel threads (calling **P_O3_L** at step 17) nor increasing the number of loop slices improves performance; in fact, performance degrades due to increase runtime overhead. Simply put, there is insufficient work to keep more than three C-90 processors busy.

4. Conclusions

In this paper, we have discussed a sequential and parallel implementation of the Pseudoknot Problem in Sisal and presented performance numbers on several multiprocessor systems. We have presented a method to write recursive programs in Sisal that preserves performance and parallelism. In the sequential code, we single-thread the data structures manipulated by the program enabling the Sisal to generate code that updates the data structures in place despite recursive calls. In the parallel code, we use a *for* expression to implement the highest branch point in the search tree, and then single-thread the data structures along each concurrent search path. The parallel program generates a sufficient number of concurrent threads to fully utilize our target machines; however, the parallelism is not dynamic. We have shown that with some insight, it is possible to write recursive, floating-point intensive parallel programs in Sisal that achieve good performance and speedups on commercial multiprocessor systems.

References

1. Cann, D. C. Retire FORTRAN? A Debate Rekindled. *CACM*, vol. 35, 8 (August 1992), pp. 81-89.
2. Freeley, M., M. Turcotte and G. LaPalme. Using Multilisp for solving constraint satisfaction problems: An application to nucleic acid 3D structure determination. to appear *Lisp and Symbolic Computations,* 1994.
3. Hartel, P. et. al. Pseudoknot: A float-intensive benchmark for functional compilers. in *Proceedings Implementation of Functional Languages Workshop 1994,* Norwich, England, September 1994.
4. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2*. Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.
5. Nico, P. and A. Park. Caching in on Sisal: Cache performance of Sisal vs. Fortran. *Proceedings Sisal '93,* San Diego, CA, October 1993.

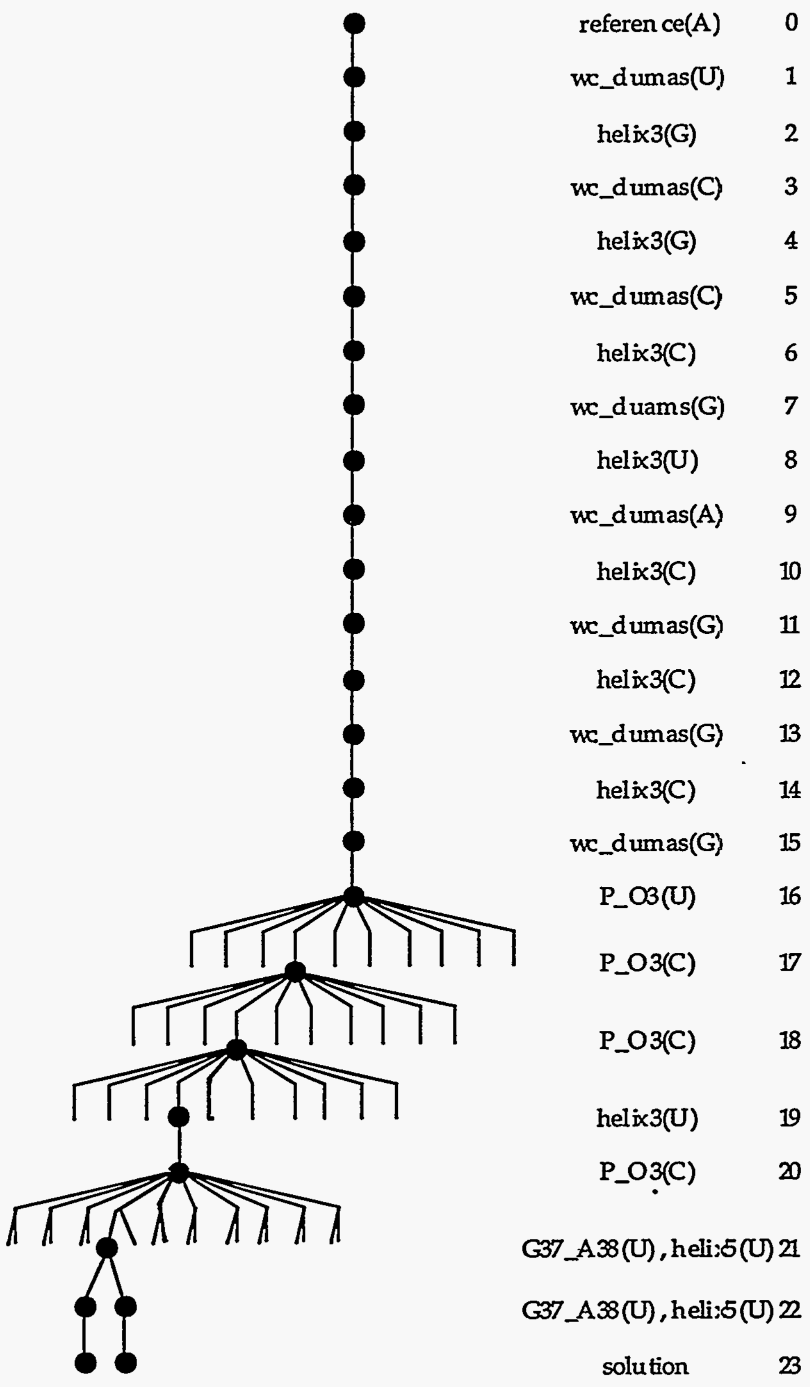

Figure 1 – A partial search tree with Sisal function calls

	Compile Time	Execution Speed/Memory			
		P1	P2	P3	P4
C	696.0 + 7.3 s	100 knts			
Sisal (sequential)	234.8 + 18.2 s	70.9 knts 125 KB			
Sisal (parallel)	255.6 + 19.3 s	55.0 knts 130 KB	95.5 knts 132 KB	119.3 knts 134 KB	126.5 knts 135 KB
Sisal (parallel, -ls10)	255.6 + 19.3 s	54.4 knts 130 KB	95.5 knts 132 KB	127.3 knts 134 KB	152.2 knts 136 KB

Table 1 – Performance on the SGI IRIS 340

	Compile Time	Execution Speed/Memory				
		P1	P2	P3	P4	P5
C	37.37 + 0.68 s	100 knts				
Sisal (sequential)	86.91 + 4.07 s	93.3 knts 229 KB				
Sisal (parallel)	106.19 + 6.21 s	88.9 knts 255 KB	164.7 knts 259 KB	215.4 knts 263 KB	243.5 knts 267 KB	280 knts 270 KB

Table 2 – Performance on the Cray C-90