CONF-9109276--4

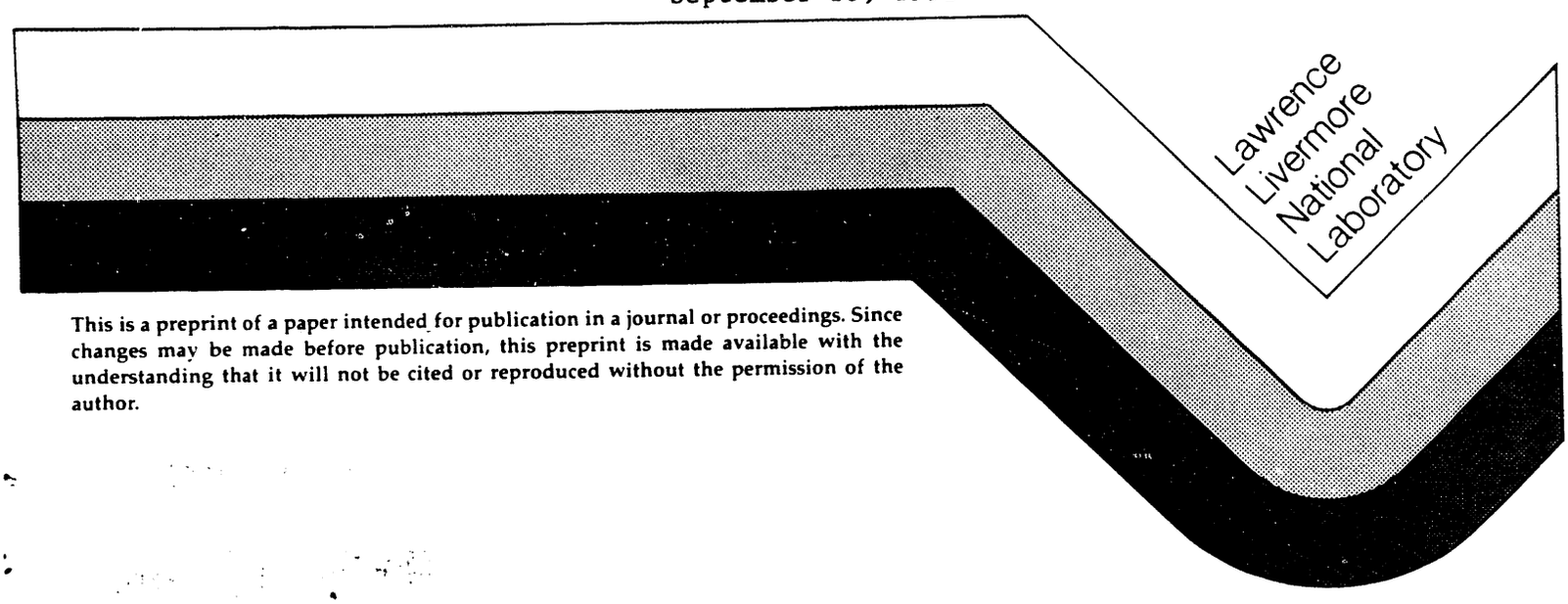
UCRL-JC-108326
PREPRINT

PROGRAMMING A REAL CODE IN A FUNCTIONAL LANGUAGE (PART I)

Christopher P. Hendrickson

The paper was prepared for submittal to
Visions of Supercomputing Conference
Santa Fe, New Mexico
September 23-27, 1991

September 10, 1991



DISCLAIMER

UCRL-JC--108326

DE92 001895

PROGRAMMING A REAL CODE IN A FUNCTIONAL LANGUAGE (PART I)

Christopher P. Hendrickson
Lawrence Livermore National Laboratory
Livermore, CA 94550

Abstract

For some, functional languages hold the promise of allowing ease of programming massively parallel computers that imperative languages such as Fortran and C do not offer. At LLNL, we have initiated a project to write the physics of a major production code in Sisal, a functional language developed at LLNL in collaboration with researchers throughout the world. We are investigating the expressibility of Sisal, as well as its performance on a shared-memory multiprocessor, the Y-MP. An interesting aspect of the project is that Sisal modules can call Fortran modules, and are callable by them. This eliminates the rewriting of 80% of the production code that would not benefit from parallel execution. Preliminary results indicate that the restrictive nature of the language does not cause problems in expressing the algorithms we have chosen. Some interesting aspects of programming in a mixed functional-imperative environment have surfaced, but can be managed.

Purpose

We are attempting to prove the feasibility of using a functional language in a real application. To this end, we are converting the physics portion of one of our main-line codes into functional form.

The first thing we wished to explore was the expressibility of the language. Since Sisal is somewhat restrictive, and since the concept of memory, or storage, does not exist for the language, we wished to understand if enough algorithms of interest could be expressed in Sisal, and if so, could they be expressed in a natural manner.

Second, we were concerned about the computational efficiency of the language. For example, in the Sisal **Scatter** function (code fragment 2), the underlying naive semantics requires copying of the array each time a new element is inserted. This is a common enough occurrence in imperative programming, and we wished to see if copy removal algorithms in the Sisal compiler worked effectively.

Third, we wished to see how much natural parallelism existed in the production code and how much would be extracted by Sisal. Assuming this amount is large and could be effectively extracted, we hope that this will lend credibility to the idea of programming in a functional manner.

Fourth, we wished to measure how long the conversion took, and how many Sisal lines of code it took to replace the equivalent Fortran lines of code.

Finally, assuming expressibility and efficiency were adequate, we wished to provide input to the Sisal group for their development of Sisal 2.0

Motivation

Parallel supercomputers have been in existence since Cray Research Inc. first brought out the Cray X-MP/24 system in 1982. However, the number of codes that have been programmed to multiprocess on this class of machines is small. At LLNL, the number can be counted on the fingers of one hand. If we believe that grand challenge problems exist[1], then these problems, which require seven orders of magnitude or greater in compute speed and a like increase in on-line storage, can only be run on massively parallel machines. Data parallel machines seem to be amenable to programming using imperative languages such as Fortran90. Here, synchronization of access to data is not an issue, as data is accessed in a synchronous manner.

With respect to asynchronous parallel machines, synchronization of data access is of paramount importance, and is the hardest part of parallel programming to get correct. We believe it will be difficult, if not impossible, to write correct massively parallel programs in imperative languages such as Fortran, C or Ada. We believe that functional languages such as Id Neuveau[2] and Sisal[3] are the languages of the future for programming asynchronous computation on massively parallel machines.

There are several benefits in writing a program in Sisal, the most significant of which is that once you have the program running correctly in a sequential manner, you only need to adjust a parameter to tell the Sisal system how many processors to run on, and the parallel program is guaranteed to run correctly. There is no way to program a race condition into the problem. Because of the functional nature of the lan-

guage, Sisal guarantees that the problem will run in a determinate manner. If the problem runs indeterminately, we will take it to the Sisal Compiler Group since only a compiler bug can generate indeterminate results.

In Sisal, a program is written so that parallelism is implicit, and sequential code is explicit. This is in contradistinction to imperative languages such as Fortran which support the von Neumann computational model. Here, the code written is implicitly sequential and must be explicitly made parallel. Since the programmer is writing code for a massively parallel system, it would seem appropriate to have a language implicitly support the hardware paradigm, and to require that any sequentiality (hopefully a minor part of the calculation if Amdahl's law is to be bested), be explicitly stated. This tends to reduce the number of lines of code when one compares fully parallel Fortran and Sisal implementations, and consequently to reduce the number of programming errors.

An interesting side benefit is that the Sisal compiler does not need to search for parallelism, nor check to see if the parallelism found can be safely exploited. All statements are capable of being safely executed in parallel, and the compiler's job is to see if there are vector or parallel methods of execution for the remaining code. An example of this, which will be discussed later, is the scatter-gather operation.

Finally, the major reason for attempting the code conversion to Sisal at this time is that the current version of Sisal seems to have solved the array copy problem[4][5]. In addition, the compiler is producing vector and scalar code which has performance comparable to the best Fortran compilers[6].

What is Sisal

The current version of Sisal, version 1.2, is not a true functional language, since it does not support higher order functions. In all other aspects, it is functional. One can think of the language as transforming inputs to outputs, in the same manner as mathematical functions. All expressions return results (note the return of two arrays by the called function, a characteristic feature of the Sisal language):

```
r,s := foo(y,z)
```

Side-effects are not allowed. For example, arrays y and z are passed by value to **foo**, and will not be changed in the context of the called routine. (The compiler will determine if arrays y and z indeed need to be copied, or can be passed by reference). Another example of a side-effect is the update by the called function **foo** of a variable accessed through **common**. Sisal has no concept of persistent state. For example, a flag cannot be toggled by one routine and read by another without the flag itself being passed as an argument to the second routine.

Sisal exists on a myriad of machines— Alliant, Cray-2, Cray X-MP, Cray Y-MP, Encore, Macintosh, Manchester Data Flow Machine, Sequent, Silicon Graphics, VAX, etc. The language is compiled to C as an intermediate language, which is then compiled in native mode to the target machine.

We need to emphasize that Sisal is a research language, and as such has few of the bells and whistles of a full production language. For example, there is no **case** statement in Sisal, so one has to make do with a sequence of **if-then-else-end if** expressions. Language development is continuing, both at LLNL and at Colorado State University, and a fully functional version of the language, Version 2.0, is currently in the definition phase[7].

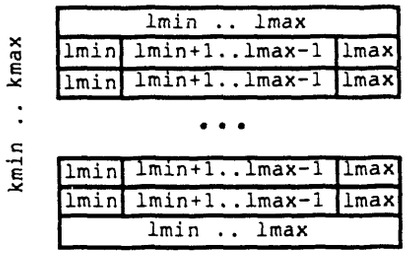
A flavor of the language

Code fragment #1, shown at the end of this paper, is an excerpt from the SIMPLE code[8] and shows how one might write a computation in both Sisal and Fortran. This calculation updates the energy for the entire internal mesh and is completely parallel if one assumes the two function calls are side-effect free. Sisal will automatically extract the parallelism since calls to Sisal functions are guaranteed to be side-effect free. Fortran may also extract full parallelism if interprocedural analysis shows the called functions to be side-effect free.

In the Fortran code, an implicit assumption is that all variables exist in **common**, and that untouched elements of the energy array will remain as they were. Note that energy(k,l) is computed, and then changed, based on some criterion. Also, note that the order of the **do** is opposite to that of Sisal, and that the loop indices could be interchanged with no change in the shape of the energy array.

In the Sisal code, an implicit assumption is that all variables will be passed as arguments, and that a new value for the energy will be returned. This array is completed in the **in**-expression portion of the function. One cannot make multiple assignments to a single variable name as in Fortran, but the **if** expression handles this nicely. A common theme in Sisal programming is that arrays must be built in their entirety. We use the **array_addh** and **array_addl** constructs at the bottom of each loop to fill in the upper and lower elements. The inner loop returns an array to which the first and lmax elements are appended. When the outer loop is completed, the

entire first and kmax rows are appended. This insures that the returned array is of size (kmin:kmax, lmin:lmax). As an aside, if the array indices were interchanged, the array returned would be of size (lmin:lmax, kmin:kmax), so the order of the indices in the loop header determines the shape of the array.



Code fragment #2 is a scatter function. This calculation updates an array with input values under the control of an index list. This code can typically be vectorized by a good compiler where vector hardware exists for the scatter function. The code is also parallel, and safe, if we assume that the index list is unique. Otherwise indeterminate execution can occur.

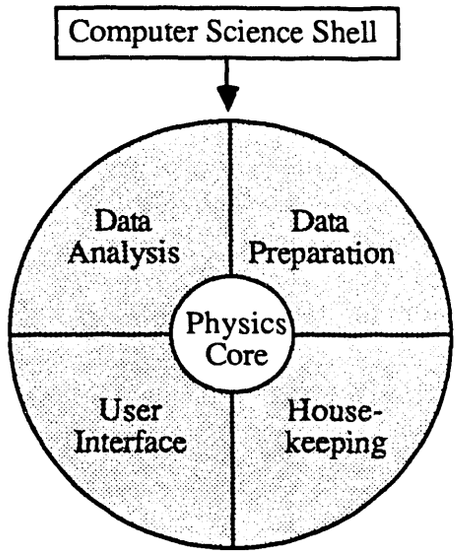
In the Fortran code, an implicit assumption is that all untouched elements of the target array will remain as they were. Therefore, returning with no action accomplished if the index list has zero length is fine. Otherwise, the array is updated in place.

In the Sisal code, an implicit assumption is that the target array will be passed in and a result array returned. The code fragment shows the syntax of the Sisal sequential loop. If the index list is empty, the loop is not executed, and the initial value of A is returned. Semantically, each iteration of the loop discards the old value of A and returns a new copy of the entire array with the single element replaced. In practice, copy removal analysis is done which shows that the array updates can be done in place, so no extra copying is needed.

This is also an example of where a production quality Sisal compiler could analyze the seemingly sequential code fragment and determine that it could indeed be vectorized, subject to a determinacy condition of the index list having unique elements.

The Mixed Language System

The Sisal Group of the Computing and Mathematics Research Division and the Compiler Group of the Livermore Computer Center have collaborated in creating a mixed language system which allows Fortran and C to call Sisal routines, as well as Sisal to call Fortran and C routines. Until now, all programming had been done completely in Sisal, except for the interface to functional mathematics routines such as **sqrt**. Over the years, it has become clear that convincing a code developer to completely rewrite his/her code in a new language was an impossible task, and was probably inappropriate as well. Most codes are actually code systems, which adhere to a model we call the CORE/SHELL model.



In this model, perhaps 20-30% of the lines of code are physics, that is, are computationally intensive, while the rest are usually scalar (perhaps parallel), and are largely concerned with state manipulation and other computation executed for side effects. We felt that the physics could be isolated via an interface from the rest of the code system, and the Sisal programming could be done in the smaller potion of the code. The production code, for example, a small one as our codes go, has 10K lines of physics and 20K lines of "other". Some of the "other" lines of code read in material properties, read dumps made from previous iterations, run other processes that provide information to the physics modules, and the like. These operations are very amenable to programming in an imperative language, and should remain in that milieu.

A second, and very important reason for providing the interface is that Fortran routines could be rewritten in Sisal and debugged one at a time, greatly simplifying the effort to debug the new code. The "big bang" approach was not necessary.

The final reason for providing the interface is that the production code system contained a rich set of utilities and other service routines that we did not want to have to duplicate.

The interface for calling Sisal from imperative routines requires that a descriptor be passed to the interface describing the bounds of the array and information as to whether the array needs to be transposed before being passed to Sisal. Standard Fortran multi-dimensional arrays are column major, and Sisal arrays are row major, so copying is typically done. Fortran allows arrays to be a larger physical size than is actually used, so the Sisal system needs to know the actual size as well as the logical size when multi-dimensional arrays are copied. Scalar data types do not require descriptors. The order of the input variables must be preserved in the Sisal routine. Output variables are listed at the end of the argument list, and again must match the order of the variables returned by the Sisal function.

From Fortran:

```
      call sisfunc(  a, adesc, i, x,
     1               r, rdesc, k    )
```

Write in Sisal:

```
function sisfunc( a: TwoD ;
                  i: integer;
                  x: real
          returns  TwoD, integer)
```

The interface for calling Fortran from Sisal is similar. The Fortran routine must be a function or be called like one from Sisal. If the input variables are scalars, and the function returns a scalar, the call is identical to Fortran.

```
y := sin(x) ;
```

If the input variables are multi-dimensional arrays, the caller must insure that the input arrays are the appropriate size, shape and majority.

```
y := foo(fixup(x)) ;
```

The Fortran function may also return an array. That is, the Fortran **subroutine foo**(x, y) where y is the result, is mapped into Sisal as:

```
y := foo(x, ydesc) ;
```

The array descriptor ydesc contains the same information as in the above case of calling Sisal from Fortran.

All parallel processing is linked into the microtasking library, and vectorization is handled by the C compiler after the Sisal program has been compiled to C. Thus, we expect to get both parallel and vector execution for our program.

What we've accomplished

We have developed a reasonably robust mixed language programming system that runs on the X-MP, Y-MP and Cray-2 under NLTSS and UNICOS.

We have programmed two major portions of the physics of the subject code in Sisal. This encompasses about half of the physics lines of code. The time for the writing and debug has, so far, been about 2 man/months of effort. To complete the physics core should take about another 3 man/months.

	Sisal	Fortran	Status
Module 1 -	2698 lines	2818 lines	debugged
Module 2 -	2783 lines	2338 lines	written

Module 1 has been integrated into the production code and runs correctly, giving identical answers to the Fortran version. Debug and integration of Module 2 will begin in early November.

The line counts for Fortran and Sisal programming have been essentially the same. The elevated numbers for Module 2 are due to programming two versions of a major sub-module, one optimized for vector performance and one for parallel performance. Some interesting aspects of this exercise will be discussed later.

What remains

We need to move the production code to UNICOS from NLTSS. We hope this will be a trivial exercise.

We need to complete the programming of the rest of the physics modules, which include some fairly difficult pieces of physics.

We would like to investigate whether the use of imperative functions can allow Sisal routines to safely and effectively manipulate state. For example, flags can be toggled if the flags are under the control of Fortran routines called from Sisal. This may simplify some of the long argument lists that we have had to use.

We need to gather performance data to see what tunings are necessary to optimize both the parallel and vector performance.

What we've learned so far

Expressibility has not yet been a problem. Some of the idiosyncrasies of the language, such as the requirement to build entire objects, make loops more verbose than in Fortran, but when one becomes accustomed to thinking in this manner it is easier to write bug-free code. For example, the first 2700 lines of Sisal code contained (so far) only 5 programming errors, three of which were easily found by turning on array bounds checking.

The imperative feel of the syntax, such as in the **pdvwork()** calculation, allows for the use of temporary variables. Since the language allows only a single assignment to a variable name, this use of temporary variables is safe. The language lends itself naturally to expressing the same kind of formulae as Fortran was originally designed for.

The ability to live inside a complete production code system means that the programmer has access to all the capabilities of that code system. For example, in writing debug edits, we were able to hook into the existing edit system with little effort. The exception handling system was also used. All the start up and shut down routines and all the routines for graphics were immediately available. And finally, the ability to run a complete problem and compare answers against the production code was a real morale builder.

Although there have been few bugs, debugging in this research compiler has been very difficult, since the compiler generates obscure C code, bearing little resemblance to the program that generated it. Symbolic debugging is all but useless. This is not a condemnation of the language, but of its experimental nature. A native mode compiler would not have these problems. In fact, a production quality compiler should have a level of optimization where the statement order written by the programmer has been preserved. The next paragraph emphasizes this need.

It is trivial to write parallel Sisal code, and difficult to write sequentialized code. Since the language generates dataflow graphs where computation ordering is driven by the availability of data, the order of computation may differ greatly from the order in which statements are written down. An example where this occurred was in the development of debug edits. An edit is a side-effect, in that is is executed for the effect it produces, not as a transformational calculation. One can fudge by having the edit routine return an integer, which can be passed to the next edit call, effectively serializing the production of edits, insuring they appear in the correct order. However, if the only data dependencies occur because of these serializing integers, these calls may all be pushed to the end of their module, or in the case where extensive subroutine in-lining is done, to the end of the whole program. Thus the edit is not taken where the calculation is made. Because the language requires single assignment to variable names, the issue is not an important one, unless an arithmetic exception happens as a result of some calculation.

As we show in the code fragments 3 and 3a, exception handling is another example of the difficulty in serializing a computation. The function **BadDensity** will do some cleanup, write a log entry and terminate the problem. It is never expected to return from the call. If the compiler is free to reorder this "computation", it may place it after the calculation of the reciprocal value, and a floating point exception which we hoped to trap will occur. Thus we need to insure that the calculation does not proceed further when a negative or zero volume is detected. The exception call, again a side effect, must be inextricably bound to the detection of the error, and not shunted to a later part of the subroutine. Since, in Sisal, both branches of the **if-then-else-end if** expression must return the same type of value, the solution to this problem is a bit ugly in that the function **BadDensity** must be declared to return a multi-dimensional array, and the subsequent calculation of the density be placed in the **else** arm of the **if**. This "orders" the calculation correctly.

An example of the dependence of the shape of the returned array on the order of the index calculations in the loop header is given in code fragments 4 and 4a. These fragments compute the two dimensional linear recurrence:

$$a_{k,l} = a_{k,l-1} * b_{k,l} + c_{k,l}$$
$$a_{k,0} = 0.0$$

Fragment 4 is optimized for parallel performance and returns a two-dimension array that has the same shape as the arrays **b** and **c**. Fragment 4a is optimized for vector performance and returns a two-dimension array that has a shape transposed from the arrays **b** and **c**. The associated Fortran code is provided. For a code developer, the method used for constructing arrays causes a good deal of heartburn, and can lead to computational inefficiencies if the array needs to be transposed to conform to the rest of the code data base. The transpose could be avoided, but at the cost of permuting the k,l indexing scheme and risking the introduction of errors, since the indexing scheme no longer matches the difference equation. Fortunately, this is to be fixed in Sisal, Version 2.0

Credits

The work on the Sisal Compiler and the Fortran-Sisal and Sisal-Fortran interface was done by David Cann of the Computing Research Group. Linda Stanberry ported the Sisal compiler from Unicos to NLTSS. Nancy Werner modified the run-time system for NLTSS, adding hooks to use the Fortran microtasking library. Both Nancy and Linda are with the Livermore Computer Center. Thanks also to Steve Hornstein, A Division, the code physicist on the production code, for his help with the production code system.

Acknowledgement

This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

[1] Proceedings of the Conference "Grand Challenges to Computational Science", held at Molokai, Hawaii - Jan 3-6, 1989, in Future Generations Computer Systems, North-Holland, Vol 5, Nos 2&3, September 1989.

[2] R. S. Nikhil. ID version 90.0 reference Manual, Computation Structures Group Memo 284-1, July 1990, revised September 1990.

[3] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language: Reference Manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[4] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD Thesis, University of California at Davis, Computer Science Department, Davis, CA, 1987.

[5] D. C. Cann. *Computational Techniques for High Performance Applicative Computation.* PhD Thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[6] D. C. Cann. *Retire Fortran? A Debate rekindled.* UCRL - JC - 107018, Lawrence Livermore National Laboratory, Livermore, CA. To appear in Supercomputing '91

[7] R. R. Oldehoeft, D. C. Cann, W. Böhm, J. T. Feo, and D. H. Grit. Sisal Reference Manual *language version 2.0.* Technical Report UCRL-JC-104008, Lawrence Livermore National Laboratory, Livermore, CA, December 1988.

[8] W.P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The Simple Code.* Technical Report UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, February 1978.

Code Fragments

Code fragment #1 - PdV work calculation from the SIMPLE Code

```
     subroutine pdvwork
      common / bounds / kmin, kmax, lmin, lmax
      common / main /   energy, dtau, p, q, temp, rho
      dimension energy(kmax, lmax), dtau(kmax, lmax), p(kmax, lmax),
     1          q(kmax, lmax), temp(kmax, lmax), rho(kmax, lmax)

      do 100 l = lmin+1, lmax-1
        do 100 k = kmin+1, kmax-1
          energyVal   = energy(k,l) - (p(k,l) + q(k,l))*dtau(k,l)
          tempVal     = tempCal(energyVal, rho(k,l), temp(k,l))
          phat        = eos(tempVal, rho(k,l), 1)
          energy(k,l) = energy(k,l) - (0.5*(phat+p(k,l))+q(k,l))*dtau(k,l)
          if ( energy(k,l).lt. 0.0 )  energy(k,l) = 0.0
 100  continue
      return
      end
```

```
 function pdvwork(bounds: ProblemSize ;
                  energy, dtau, p, q, temp, rho: Twod
          returns Twod                                    )
   let
     kmin := bounds.kmin ;
     kmax := bounds.kmax ;
     lmin := bounds.lmin ;
     lmax := bounds.lmax ;

     em := for k in kmin+1, kmax-1
       er := for l in lmin+1, lmax-1
         energyVal := energy[k,l] - (p[k,l] + q[k,l])*dtau[k,l] ;
         tempVal   := tempCal(energyVal, rho[k,l], temp[k,l]) ;
         phat      := eos(tempVal, rho[k,l], 1) ;
         eV1       := energy[k,l] - (0.5*(phat+p[k,l])+q[k,l])*dtau[k,l] ;
         returns array of if eV1 < 0.0 then 0.0 else eV1 end if
       end for;
        returns array of array_addl(array_addh(er, energy[k,lmax]),energy[k,lmin])
     end for;
   in
     array_addl(array_addh(em, energy[kmax]), energy[kmin])
   end let
 end function
```

Code Fragments

Code fragment #2 - Scatter data movement

```
      subroutine scatter(target, indexlst, length, source)
       dimension target(1), indexlst(1), source(1)
       if ( length .gt. 0 ) then
         do 100 i = 1, length
           target(indexlst(i)) = source(i)
  100    continue
       endif
       return
       end
```

```
function scatter(target: OneD ; indexlist: OneI ; source: OneD returns OneD)
  let
    length:= array_size(indexlist) ;
  in
    for initial
      i := 0 ;
      a := target ;
    while i < length repeat
      i := old i + 1 ;
      A := old A[indexlist[i]: source[i]] ;
      returns value of A
    end for
  end let
end function
```

Code Fragments

Code fragment #3 - Exception Handling - Incorrect

```
global BadDensity(zoneList: TwoI returns integer) ;

  badZones, numberBadZones :=
    for k in kmin+1, kmax-1 cross l in lmin+1, lmax-1
      theVal := if tau[k,l] > 0.0 then 0 else 1 end if ;
      returns array of theVal
              value of sum theVal
    end for ;

  dummy := if numberBadZones ~= 0 then
             BadDensity(badZones)
           else
             0
           end if ;

  density :=
    for k in kmin+1, kmax-1 cross l in lmin+1, lmax-1
      returns array of 1.0/tau[k,l]
    end for ;
```

Code fragment #3a - Exception Handling - Correct

```
global BadDensity(zoneList: TwoI returns TwoD) ;

  density := if numberBadZones ~= 0 then
               BadDensity(badZones)
             else
               for k in kmin+1, kmax-1 cross l in lmin+1, lmax-1
                 returns array of 1.0/tau[k,l]
               end for
             end if ;
```

Code Fragments

Code fragment #4 - Two dimensional linear recursion, optimized for parallel execution

```
c       Fortran sequential-sequential
c       output array shape is [1..kmax][1..lmax]

      do 100 k = 1, kmax
        a(k,1) = c(k,1)
        do 110 l = 2, lmax
          a(k,l) = a(k,l-1)*b(k,l) + c(k,l)
  110   continue
  100 continue

%       Sisal parallel-sequential
%       output array shape is [1..kmax][1..lmax]

  a :=
    for k in 1, kmax
      ar :=
        for initial
          l := 1 ;
          a := c[k,1] ;
          while l < lmax repeat
            l := old l + 1 ;
            a := old a*b[k,l] + c[k,l] ;
          returns array of a
        end for
      returns array of ar
    end for ;
```

Code fragment #4a - Two dimensional linear recursion, optimized for vector execution

```
c       Fortran sequential-vector
c       output array shape is [1..kmax][1..lmax]

      do 100 k = 1, kmax
        a(k,1) = c(k,1)
  100 continue

      do 110 l = 2, lmax
        do 110 k = 1, kmax
          a(k,l) = a(k,l-1)*b(k,l) + c(k,l)
  110 continue

%       Sisal sequential-vector
%       output array shape is [1..lmax][1..kmax]

  emptyArray := array TwoD [] ;
  firstRow   := for k := 1, kmax
                  returns array of c[k,1]
                end for ;
  a :=
    for initial
       l := 1 ;
       a := array_addh(emptyArray, firstRow) ;
     while l < lmax repeat
       l := old l + 1 ;
       nextRow := for k in 1, kmax
                    returns array of old a[k,l-1]*b[k,l] + c[k,l] ;
                  end for ;
        a := array_addh(old a, nextRow)
     returns value of a
    end for ;
```