The Sisal Project: Real World Functional Programming [⋆]

Jean-Luc Gaudiot[1], Tom DeBoni[2], John Feo[2], Wim Böhm[3], Walid Najjar[3], and Patrick Miller[4]

[1] University of Southern California
Department of Electrical Engineering - Systems
Los Angeles, California
`gaudiot@usc.edu`

[2] Lawrence Livermore National Laboratory
Computer Research Group
Livermore, California
`{deboni,feo}@llnl.gov`

[3] Colorado State University
Computer Science Department
Fort Collins, Colorado
`{bohm,najjar}@cs.colostate.edu`

[4] CaeSoft Development
Tracy, California
`patmiller@sisal.com`

Abstract. Programming massively-parallel machine is a daunting task for any human programmer and parallelization may even be impossible for any compiler. Instead, the functional programming paradigm may prove to be an ideal solution by providing an implicitly parallel interface to the programmer. We describe here the Sisal project (Stream and Iteration in a Single Assignment Language) and its goal to provide a general-purpose user interface for a wide range of parallel processing platforms.

1 Introduction

The history of computing has shown shifts from explicit to implicit programming. In the early days, computers were programmed in assembly language, mostly with the purpose of utilizing the available memory space as effectively as possible. This came at the cost of obscure, machine-dependent, hard to maintain programs, which were designed with high programming effort. Fortran

[⋆] This research is supported in part by ARPA grant # DABT63-95-0093 and NSF grant 53-4503-3481 was introduced to make programming more implicit, portable and less machine-dependent. With the advent of massively parallel computers and their promise of hundreds of gigaflops, we have seen a return to the explicit programming paradigm. Using for example C with explicit message passing library routines as "machine language," people attempt to utilize the available processing power to the largest extent, again at the cost of high programming effort, machine-dependent, and hard to maintain code. A compiler for an implicitly parallel programming language alleviates the programmer from the task of partitioning program and data over the massively parallel machine.

It is our view that explicit parallel programming is a transition stage in the evolution of parallel computing and that implicit parallel programming languages will eventually become the norm as did high-level languages in the sequential paradigm. This will result in a tremendous improvement in *programming quality* in terms of programming effort, readability, portability, extendability and maintainability of parallel code. Another consequence will be the accessibility of parallel programming to a wider public that would make use of a wide spectrum of parallel computers: from a few processors on a chip to several thousand processor-machines.

Functional programming [6] is an alternate programming paradigm which is entirely different from the conventional model: a functional program can be recursively defined as a composition of functions where each function can itself be another composition of functions or a primitive operator (such as arithmetic operators, etc.). The programmer need not be concerned with explicit specification of parallel processes since independent functions are activated by the predecessor functions and the data dependencies of the program. This also means that control can be distributed. Further, no central memory system is inherent to the model since data is not "written in" by any instruction but is "'passed' from" one function to the next.

Sisal (Stream and Iteration in a Single Assignment Language) [22] is such a functional language which was originally designed by collaborating teams from the Lawrence Livermore National Laboratory, Colorado State University, the University of Manchester and Digital Equipment Corporation. The goal of the project was to design a general-purpose implicitly parallel language for a wide range of parallel platforms.

The goal of this paper is to describe the last phases of this project as we are currently undertaking them. In section 2, a short tutorial will present the basic principles of Sisal. An early compiler implementation for shared memory systems is described in section 3. Sisal 90 and its foreign language interface is introduced in section 4. We turn our attention to Distributed Memory implementations in section 5, while section 6 introduces implementation of multithreading principles. Section 7 concludes.

2 The Sisal language: a short tutorial

Sisal is a functional language that offers automatic exploitation and management of parallelism as a result of its functional semantics. In Sisal, and all functional languages, user-defined names are "identifiers" rather than variables, and they refer to values rather than memory locations. The values produced and used in a Sisal program are all dynamic entities, and their identifiers are defined, or bound to them, only for the duration of their existence in an execution. This is the dynamic of the data flow graph, in which graph nodes are operations, and values are carried on the arcs connecting the nodes. The extent of the existence of a value is the set of arcs on which it travels between the point of its definition and the point of its final consumption. The values that are defined by the graph arcs may or may not have names assigned to them within a program.

All Sisal expressions and higher-level syntactic elements evaluate to and return values based solely on the values bound to their formal arguments and constituent identifiers. This eliminates any possibility of side effects, and allows much richer analyses of program by the compiler than is typically the case for imperative languages.

To best illustrate these points, consider the following brief code fragment. It is written in Sisal 1.2, the language currently accepted by the Optimizing Sisal Compiler. The Sisal language is undergoing expansion and refinement, as discussed in other sections, but the syntax of version 1.2 will suffice for this example.

```
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
```

The first two statements define type names for arrays. Note that no sizes are provided; all Sisal aggregate data instances are dynamically created, resized, and de-allocated at runtime. Only the dimensionality and element types are relevant to the type specifications. The header for function "generate" shows that one integer argument, "n", is expected, and two unnamed values will be returned. The returned values are two dimensional arrays of single precision reals, but again, only typing and not sizing is specified. Names can be bound to these returned values at the site of invocation of function generate if the programmer wishes. An invocation of a function is semantically equivalent to the reproduction of the function code at that site, with appropriate argument substitution.

This equivalence, called "referential transparency" is a fundamental property of functional languages, and is responsible for the strengths of the Sisal language. This strength lies in a simplified analysis process for the compiler. Functions can run in parallel if no data dependency exists between the functions. Functions with equivalent inputs will **always** return equivalent values.

All Sisal expressions, including whole functions and programs, evaluate to value sets. In the above case, the function evaluates to two arrays, which are the values of the expression contained in the function definition. The for-expression shown is a loop construct, which is an indicator of potential parallelism to the Sisal compiler. This loop has an index range defined as the cross product of two simpler ranges. This means that the body of the loop will be instantiated as many times as there are values in the index range, in this case n*n, and each body instantiation will be independent, since no data dependencies exist among them. The set of independent loop bodies can be executed in parallel or not, based on the compiler's and the runtime system's analyses of their costs, as well as on options specified by the programmer.

The appearance of the names "t1" and "t2" within the body of the loop should not be considered a reuse of these names in the sense of the reassignment of a variable in an imperative program. Instead, the names are used to define the computation in the loop body, and in fact these names will likely have no real existence within the executing program. The important point here is that each instance of the loop body, containing specific values for i and j, will independently compute specific instances of the values defined as real(i)*real(j) and real(i)/real(j); then all these separate values will be gathered together into a pair of arrays and returned. The positions of the values in the result arrays are determined by the loop's index ranges, as are the overall size and dimensionality of the returned arrays. In this case, two two-dimensional arrays are returned, with index ranges from 1 to n in each dimension. The use of loop-temporary names is optional, and the return-clause above could be rewritten as:

```
returns array of real(i)*real(j)
        array of real(i)/real(j)
```

with no change in the ultimate results. The loop body, then, would appear to be empty, but in fact, the language treats the expressions in the array-of clause as anonymous temporaries.

Further syntactic elements of the Sisal language include let-in statements, which allow for name definition and use; if-statements, which allow conditional name definition; record and union types, which allow for flexible data aggregation; streams, which allow for producer-consumer computations; and sequential loops, which allow true iteration, with specified data dependencies existing between iterations. I/O in Sisal is performed by passing inputs as arguments to, and receiving outputs as the results of, the outermost function. The values used for inputs and returned as outputs obey a syntax called "Fibre", which allows the demarcation of dynamically sized aggregates.

The Optimizing Sisal Compiler translates source programs into executable memory images, including the runtime system components required to automatically manage memory, tasking, and I/O. The amount of parallelism to be exploited by a program can be controlled by user options, and once compiled, a program can be executed by any number of worker processes, by way of a single runtime parameter. Similarly, compiler optimization behavior and runtime performance can be observed and controlled by options applied at various points during compilation and execution.

3 An Early Implementation: The Optimizing Sisal Compiler

Early implementations of the Sisal language were basic proofs of concept. Various interpreters have been implemented, like DI[36], TWINE[23], and SSI[24] but for greatest execution speeds, a compiled code was needed. Sisal was ported to novel architectures like the Manchester Dataflow Machine [7], but complete acceptance for the language required porting to newly emerging shared memory parallel machines then coming to market (HEP[1], Encore, Sequent[27], Cray).

3.1 Update in place and copy elimination

Several key obstacles emerged. The first of these came from Sisal's semantic concept of making "copies" to preserve single assignment and referential transparency. A fragment like

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

returned [1: 1,2,3], [1: 1,999,3]. The original value of A had to be preserved, so a copy was made to enable the replacement.

Consider instead swapping two elements of an array. In Sisal, one would write something like:

```
C  := array[1: 1,2,3,4,5];
% Read as E is identical to C
% Except index 3 has C[4] in it
% Index 4 has C[3] in it
E  := C[3: C[4]; 4: C[3] ];
```

The *semantics* (not the implementation) of Sisal calls for making a copy of the array to make the first replacement, and making another copy for the second replacement. A FORTRAN programmer would never do this, instead they would write:

```
ITEMP = IARRAY[3]
IARRAY[3] = IARRAY[4]
IARRAY[4] = ITEMP
```

This program has no (array) copies and is done in place. Similarly, many 'in-place' algorithms that are efficient in space and time have been designed in imperative languages that would run poorly if all data structures had to be copied. Clearly there was room for improvement[38]. Consider the original Sisal swap broken down into individual steps:

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

When the array replacement is done on line 3, the value in "C" is dead. When the replacement is done, the C value can be thrown away. Instead of making a copy for D and throwing C away, we can safely use C's container instead. Similarly, the second replacement on line 4 is the last use of D, so E can use D's container. This analysis is simplified because of the functional semantics of Sisal. The Optimizing Sisal Compiler (OSC) makes heavy use of "update-in-place" and copy elimination analysis to eliminate many unnecessary copies[9]. In its simplest sense, update-in-place migrates reader operations before writers. Here, the C[3] and C[4] readers were moved before the replacement operations in order to maximize the chances that C would be a "dead" value for the update.

3.2 Build in place

Other important optimizations had to be developed [30]. For instance, many functional programs work on pieces of a large structure and then "Glue" the computed fragments together. For instance:

```
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);
LIIIR := array_addh(LIII,R);
```

Semantically, this says: 1) Build piece L, 2) Build piece R, 3) Build size n-2 array III, 4) Allocate T1, a size n-1 array, 5) Copy L and III into T1, 5) Allocate LIIIR, a size n array, 6) Copy LIII and R into LIIIR This seems to require two allocations and a two large data copies. OSC introduced the idea of "BUFFERS" and persistent memory to the BACKEND of the compiler – leaving the frontend unchanged. Using a buffer system, the same operation proceeds as follows:

```
Build Buffer LIIIR of size n
Compute L and put in LIIIR[1]
Compute R and put in LIIIR[n]
Compute III in LIIIR[2]...LIIIR[n-1]
```

This trick can be played even if the left and right pieces are loops. The beauty of this "Build-in-place" system is that memory can be preallocated and parallel computations can simply stick values where they belong – even if the original computation parts were from distant parts of the computations.

3.3 Reference Counting Optimization

We have seen that we can take advantage of an object ending its life just as we would otherwise need to copy. Reference counts were introduced to help know when: 1) a value can be updated in place and when 2) a value's memory can be recycled. Reference counting can be a very expensive operation on sequential machines – on parallel machines it is much worse!!! Parallel reference counts must be updated in a critical section. This operation keeps banging on locks every few operations, swamping the machine. Luckily, programs tend to have simple patterns of use for aggregate values and OSC can cleverly eliminate[35] nearly all reference counting in a program through lifetime analysis and operation merging.

3.4 Vectorization

On vector machines all the speed advantages come from routing array operations through temporary vector registers. OSC has fine control of loop placement so that reader/writer chains can be established. In imperative languages, this generally requires very careful writing of loops in order to clearly establish vector relationships between loops. The semantics of Sisal's underlying dataflow representation make loops easy to move and so OSC can vectorize extremely well[11].

3.5 Loop Fusion, Double Buffering Pointer Swap, and Inversion

On scalar and scalar/parallel machines, loop overhead and memory fetch time tends to dominate computations. OSC can accommodate these machines by applying aggressive loop fusion. Fusion can rewrite loop code like

```
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;
```

into

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

eliminating the generation of two temporary arrays and setting values that can stream into internal registers from the cache. FORTRAN90 has similar semantics for its array operations: X = A*2 + B*3. Sisal's OSC compiler can implement this more efficiently than a FORTRAN90 compiler because FORTRAN must know absolutely that neither A nor B is aliased to X. Sisal's functional semantics insure that a left-hand-side of a definition is **never** an alias for a right-hand-side.

A typical scientific computation proceeds as follows:

```
for initial
   A := start_values()
while not done(A) repeat
   A := time_step(old A)
...
```

Here, a new version of A the same size is generated at each time step. A naive implementation of Sisal would allocate a new buffer for each time step and throw away the old even though it was the right size. OSC notices this and initially allocates a buffer outside the loop and pointer swaps the original and secondary buffers.

Consider a 1D smoothing function that averages values using a three point stencil – X[i] = (A[i-1] + A[i] + A[i+1])/3.0 At the endpoints a two point stencil is used instead. This is most easily expressed as:

```
X :=  for i in 1,n
       v := if i = 1 then (A[1]+A[2])/2.0
            elseif i = n then (A[n-1]+A[n])/2.0
            else (A[i-1] + A[i] + A[i+1])/3.0
            end if
      returns array of v
      end for
```

The if-tests appear to introduce a large overhead and to inhibit parallelism, vectorization and pipelining. The loop can be specialized doing the boundary computations separate from the inner computations. The inner computation is simple:

```
inner := for i in 2,n-1 returns
            array of (A[i-1]+A[i]+A[i+1])/3.0
         end for
```

Now we just need to glue on the lower bound computation and the upper bound computation. We need to be careful to handle zero trip loops here! This is done by producing an *array* of for the boundary values. In zero trip case, an empty array is generated. In all other cases, an array of size 1 is generated.

```
leftbound := for i in 1,min(1,n) returns
               array of (A[i]+A[i+1])/2.0
             end for;

rghtbound := for i in max(2,n),n returns
               array of (A[i-1]+A[i])/2.0
             end for;

        X := leftBound || inner || rghtBound;
```

The max/min function calls make sure the zero trip cases are handled gracefully. The final catenation puts the results in the correct form. The catenations will actually be removed by build-in-place optimizations later in the optimization process.

4 Sisal90

The original Sisal definition has been extended and modernized. The new language includes language level support for complex values, array and vector operations, static polymorphism and type-sets, higher order functions, user-defined reductions, rectangular arrays, and an explicit interface to other languages like FORTRAN and C. See [14] for a detailed description of Sisal90 and a comparison with Sisal 1.2.

An important objective was to enhance the language definition while maintaining compatibility with the Sisal 1.2 definition. We could not delay our application work waiting for the new definition and compilers, nor disenfranchise the extant Sisal community. Additionally, enhancement rather than overhaul implies fewer changes to the backend and permits us to reuse existing software. The desired new features prompted a full rewrite of the parser and a complete rethinking of how the low-level operations had to be specified.

A second objective was to increase Sisal's appeal to scientific programmers. To this end, we adopted Fortran 90 array operations where possible, improved support for mixed-language programming, and included features, perhaps not consistent with a strict interpretation of functional dogma, that simplify the programmer's task. We do not believe that functional languages can survive on their own; however, they can play a critical support role. Most of the code in a large scientific application pertains to problem specification, termination, I/O, and fault handling. These sections are not functional and not parallel; write them in your favourite imperative programming language. However, often the computational kernel is parallel and functional. Here Sisal can play a crucial role, as it can reduce development costs, insure determinacy, and improve portability without sacrificing performance. We perceive a gradual merging of the functional and imperative programming communities where functional constructs form either a set of language extensions or an integrated core. We hope that the Sisal 90 definition will accelerate this process.

4.1 The Foreign Language Interface

Parallel programming traditionally involves the management of concurrent tasks and machine resources in addition to the specification of the computation, greatly increasing the programmer's burden. Most parallel programs are not written for parallel execution from the outset. More often, they begin as existing sequential programs, written in an imperative language, and are augmented with parallel constructs from a vendor-specific enhanced imperative language. The programmer who is assigned the task of parallelizing such a code must preserve the semantics of the program; the parallel code must execute efficiently on the parallel machine of choice and must exhibit some scalability; the code should port easily to other parallel machines, in particular new generations of the target machine, and development costs should be kept as low as possible. As these goals are contradictory, there may be no best solution. Since minimizing programming costs is an important objective, the programmer usually identifies the most computationally intensive parts of the code, and parallelizes only the parts that will provide the most gain from parallel execution. By considering only these parallelizable sections, the imperative programmer maximizes performance and minimizes development costs. The Sisal language supports mixed language programming through its Foreign Language Interface (FLI). The FLI allows Sisal programs to call or be called from Fortran or C, and to invoke existing libraries or solvers. This allows relatively easy recoding of the computational kernels of an existing code for parallelism.

The use of the Sisal FLI involves four steps. First, the appropriate level of parallelism, and the portion of the original code that contains it must be identified. The size of computational grains to be parallelized and the amount of communication they will do must be considered. There may be one identifiable code region which is appropriate for parallelization, or there may be many several, separated by sequential portions of the code. Second, the data that must be communicated into and out of Sisal must be identified. This is important, since Sisal's functional semantics require a strict separation of inputs and outputs. The mere determination of the input and output data may be nontrivial, since the imperative language may hide the data in global variables, common blocks, and aliased variables, and their use as input, output, or both, may be difficult to discern or may be situation dependent. The third step is usually easy, once the first two have been achieved; it is the translation of imperative source code into Sisal. While no automatic machanisms have been developed to do this, due to its dependency on human intelligence and information gleaned from the first two steps, it is can usually be accompished by a straightforward set of edits. The fourth step deals with the data movement between Sisal and the imperative language, and the initiation and termination of the Sisal Run Time System.

We will not address the first two steps, mentioned above, as they represent an entire genre of know-how and experimentation by themselves. Step three will require familiarity with both Sisal and the imperative language code under consideration. Since most practically exploitable potential parallelism in existing imperative codes will come from loops, they should be examined first. Sisal does quite well at slicing its parallel loops. Other sorts of parallelism, such as function parallelism (independent functions that can potentially execute concurrently) and producer-consumer parallelism (e.g. software pipelines) are not currently exploited by the Sisal compiler and Run Time System, so they can be ignored. Once a loop has been identified as a target for parallelism, it must be examined for inter-iteration dependencies. These will inhibit parallelism, and must be eliminated from the parallel loops that result from the translation step. Separate interative loops may need to be built in Sisal to handle these portions of the code. The imperative loops will often have false dependencies in them arising from the reuse of variables where no real data dependency is present. These can be eliminated by the use of loop temporary names in Sisal. Imperative loops also often have assignments with indexed array names as their targets; these mush also be leiminated, and can usually be rewritten with loop temporaries, given appropriate index arithmetic. Once the programmer is used to dealing with these exigencies, the translation process can be quick and easy. Following are two code fragments illustrating these details.

```
       temp = 0.0
       Do 100 i = M, N
        temp = temp + A(i)
        B(i) = func( A, i )
        C(i) = C(i-1) + A(i)
        A(i) = A(i)*2.0
 100   continue
```

The first loop is in Fortran. Its inputs are a scalar, temp, an array, A and an array C; its outputs are the scalar temp, and arrays A, B, and C. Note that array C has an index range apparently differing from those of A and B by one: it has an element C(M-1), while A and B may not have an element indexed less than M. The calculation of temp seems inherently sequential, but in fact it can be accomplished in a parallel loop in Sisal. Here is a translation of the above loop into Sisal:

```
temp, New_A, B, New_C :=
 for i in M, N
  A_sub_i := A[i] ;
  B_sub_i := foo( A, i );
  C_sub_i := C[i-1] + A_sub_i;
  New_A_sub_i := A_sub_i * 2.0;
 returns value of sum A_sub_i
         array of New_A_sub_i
         array of B_sub_i
         array of C_sub_i
 end for
```

The syntactic differences should be obvious, as should the simplicity of the translation between them. It should be noted that the Sisal fragment is artificially lengthened by the presence of the simple expressions in the loop body. In fact, all those expressions could be in the returns clause, which would make the Sisal loop no longer than the Fortran version. However, we find clarity to be more important than brevity, in many cases where parallelism is the goal and accuracy is at risk, so we tend to use loop temporaries, as shown above, to help make Sisal code readable, and to err on the side of of readability where style is arguable.

Step four involves building argument lists for the Sisal Run Time System to use in invoking the outermost Sisal function, and return value lists for the RTS to pass back to the invoking inmperative code. Scalar data can simply be passed in and returned without special effort, but arrays are more complicated. Arrays in C and Fortran are contiguous blocks of primitive type elements stored row-wise (in C) and column-wise (in Fortran). Arrays in Sisal are vectors of scalars or vectors, which are contiguous only in the most primitive dimension, and are stored row-wise. When passing an array between an imperative program and a Sisal function, a descriptor must be provided in addition to the array, that allows the Sisal Run Time System to correctly handle the data and mange the storage it uses. Since all data items in Sisal ("values", as opposed to "variables" in imperative languages) are dynamic, storage must be managed by the RTS. It does this well, and normally requires no help from the programmer. Mixed language programming requires extra efforts in the form of array descriptors. Each array requires a descriptor, and each descriptor contains fields for each dimension of the target array that describe that dimension's physical and logical index range, whether the data is read-only or writeable, and whether it must be transposed in passage. Fortran arrays, for example, if of more than one dimension, must be transposed by the RTS, since they are allocated in column major order in Fortran and row major order in Sisal. The descriptors are themselves small arrays which must be allocated in the imperative code, and which must be provided for each array argument and result. The provision of this information can conceivably be automated by the compiler, but at present it must be performed manually by the programmer.

In addition to the above, the Sisal Run Time System must be started and stopped at points in the imperative program that are appropriate to the parallel work that will be done. Normally, the RTS is started and stopped automatically during the execution of a pure Sisal program, but this code must be explicitly included and invoked in the loading and execution of a hybrid program. Since it is expensive in terms of CPU time to do this, it is not appropriate that it be done repeatedly within a loop that contains calls to the Sisal code. Rather, the RTS should be started once, the Sisal code invoked wherever appropriate, and the RTS should be shut down before the normal termination of the program. It costs relatively little to leave the RTS running between invocations of the Sisal code, so this method is not particularly wasteful of machine resources. The RTS is started by a simple call which contains a few of the parameters normally used in te execution of a Sisal program. These include the program heap size (the memory pool used by the RTS), the number of worker processes to be used (the amount of parallelism exploited).

At this point it is worth mentioning that the Sisal FLI was built as an experiment, and as such is still in a somewhat rougher state than would be desired in a production parallelization system. Its use, as documented above, can present difficulties that can effectively undo some of the advantages of applicative paralleism. For instance, the generation of the array argument and result descriptors adds to the programmer's burden. In addition, arrays of dimension greater than one will currently be copied across the FLI, a source of overhead at exectute time that is inimical to parallel performance goals. Therefore, in the work we have done with it, we have routinely used aliasing to hide the multidimensional nature of such arrays, and index arithmetic to allow arbitrary access to their elements.

In addition, we must confess that the goals of machine independence are not always met in parallel programming, and this is at least as true in mixed language programming for parallelism. It sometimes happens that the Sisal code resulting from the translation of step three, above, must be modified for performance purposes. For instance, column-wise accesses to two-dimensional arrays usually causes performenace degradation in systems containing cache memories. However, this by itself is usually no more serious a constraint than would be imposed by such system architectures during a parallel port in any other language.

Notwithstanding the problems mentioned above we believe the FLI offers two distinct advantages to the parallel programmer. First, it provides a means of rapidly parallelizing existing application codes by concentrating programmer effort where it will provide the best return. And second, it offers a developmental path for codes ranging from experimentation on cheap workstations to production on expensive supercomputers.

5 A Prototype Distributed-Memory SISAL Compiler

In this section we present D-OSC, a prototype SISAL compiler for distributed-memory machines. D-OSC is an extension of OSC[12]. A new analysis phase for loop and array distribution has been added and the code generation phase has been modified to produce C plus MPI[15] calls. The run-time system has been modified to support array distribution and communicating threads. Information needed to perform distributed memory optimizations is established by the analysis phase and provided to the code generator by decorating the appropriate IF2 nodes and edges.

The D-OSC model of execution is activation-based. A *master* process is responsible for dividing parallel loops into *slices* which will be executed by *slave* processes running in parallel. A slice is represented by an *activation record*, which contains a code pointer, the loop range, a unique loop identifier, input parameters to the slice, and destinations for values to be returned upon termination. Activation records are distributed over the machine and each processor maintains a local *activation record queue*. Upon completion of a slice, the slave process sends a completion message to the master and updates global results with locally-computed values. As a slice may contain a parallel loop, each slave can become a master and distribute its inner loop. Each processor must be able to receive a request for service from other processors, such as a read, write or allocate request. This is achieved by having a *listener* thread always active on every processor.

D-OSC is implemented in four phases, where each phase relies on the previous one.

- *Base.* This phase employs no analysis whatsoever, hence the code generated is very naive. Arrays and loops are distributed equally among processors. Message passing is used to access remote array elements. This compiler version serves as a reference for further implementations, providing useful information about the effectiveness of certain optimizations.
- *Rectangular Arrays.* The standard implementation of higher-dimensional arrays as arrays of arrays is replaced, where possible, by rectangular arrays with a single descriptor. Arrays and the loops creating or using arrays can be distributed by rows, block or columns. Not all loops are distributed.
- *Block Messages.* The reading and writing of remote array elements within certain loops is optimized by combining all the messages directed to the same processor into a single block message.
- *Multiple Alignment.* In previous phases arrays partitioning created disjoint sections of an array. In this phase *overlapping* array sections are created. This optimization reduces the number of messages passed, at the cost of using more space for the overlapping array sections.

5.1 Base Compiler

In OSC, the representation of arrays consists of an array descriptor, which contains information such as bounds, reference count, size, and other information, and a pointer to the physical array. OSC assumes a shared-memory model, and the pointers to the array descriptor provide a unique array identifier. An evident problem on a distributed-memory machine is that the descriptor pointer cannot be used as a unique identifier, since the address of the array descriptor is different for each processor. Hence a unique array identifier is created explicitly as the index in an array table that exists on each processor. The design of the array table permits a great deal of compatibility with existing array operations since the OSC concept of a unique array descriptor is preserved.

Arrays are partitioned according to the distribution of the creating loop. In the Base compiler each loop, and hence each array dimension, is distributed equally among processors. To create the unique identifier for distributed arrays, the master process that creates the loop slices, allocates the array identifier and sends it as part of the activation message to the slaves. Each slave then executes a slice in parallel and updates its local entry in the array table.

Array access in the base compiler is straight-forward. The processor that owns the array element is determined. In the base case this amounts to a simple computation involving the array size and the number of processors. If the owner is the local processor, the array element is read directly from local memory, otherwise a request message is sent to the listener thread of the processor that owns the array element. The listener thread directly performs the array access.

5.2 Rectangular Arrays

Rectangular arrays have only one descriptor per array, regardless of its dimensionality. Only one possibly remote memory access to fetch the array element is needed, where an arrays of arrays implementation of an nD array requires n memory accesses to fetch an element. With one array descriptor per array traditional distributions, such as row, block and column, are easier to implement. A disadvantage of rectangular arrays is that sub-arrays cannot be shared. However, sharing also has disadvantages since update-in-place cannot be performed, and access functions are less efficient. Another disadvantage of rectangular arrays is that ragged arrays cannot be represented.

Arrays are created using IF2 `AGather` nodes. Consider the case of a Sisal *triple cross product for* loop that returns a three-dimensional array. In the original IF2, `AGather` nodes in the result graphs of all three nested loops create arrays. In the rectangular array case, the actions that the various `AGather` nodes perform are different. The outermost `AGather` node must perform the allocation of the physical space for the whole 3D-array, and the allocation of the single array descriptor. The innermost `AGather` node fills in the elements of the array. The `AGather` node in the middle loop does not perform any action.

In the original IF2 an arrays of arrays access consists of multiple `AElement` nodes scattered over the dependence graph, each with one index input. For a rectangular array this must be transformed into one `AElement` node with all indices as input. The analysis phase identifies the *tree* of `AElement` nodes that is spanned by the output edge of a root `AElement` node and marks these nodes with information such as the level of the node in the tree and back-edges to ancestor nodes.

5.3 Block Messages

The implementation of array access operations described above is not always efficient for array references in loop bodies, as performing remote exchanges for individual elements is less efficient than performing at most one block exchange per producer-consumer processor pair. Our algorithm for obtaining block messages is a modification of the algorithm presented in [16].

5.4 Multiple Alignment

The last phase of the compiler implements the overlapping allocation of array sections presented in [17] for one-dimensional arrays. Overlapping allocation is applied to loops with *restricted affine* references as in the following loop model, where the `cj`s are constants.

```
for i in lo, hi
 returns array of f(B1[i+c1],...,Bm[i+cm])
end for
```

In the case of *single alignment*, i.e. $m = 1$, the first element of the consumer array is aligned with element $1 + c_1$ of the producer array. For the general case, the analysis phase identifies restricted affine loops, that create one-dimensional arrays while accessing elements from other one-dimensional arrays. Multiple alignment is achieved by identifying all the unaligned references required, and the maximum and minimum offsets of these with respect to the consumer index. The contiguous set of indices thus obtained is a superset of the producer array elements needed. Loops are marked *RightOverlap* and *LeftOverlap* to be used in the code generation phase to determine the upper and lower bounds for each slice.

5.5 Results

The benchmark programs used here to assess the effectiveness of the various optimization phases are Livermore loops 1, 2, 3, 6, 7, 9, 12, 21, and 24, run on a network of four workstations. Since the initial objective is to reduce communication, we measure the total number of messages exchanged - the first number in table 1, and the total volume of communication - the second number in the table.

Rectangular arrays decrease the number of messages exchanged for some of the programs that use 2-D arrays. However, sometimes the number of messages increases, as in loop 21. The reason for this is that the partitioning of the loops and arrays performed by the base compiler matches the accesses of the array elements better than the rectangular arrays implementation.

Table 1. Number of Messages, Communication Volume (4 PEs).

Program	*Type*	*Base*	*Rect Arrays*	*Block Mssgs*	*Multiple Algn*
ll1	1D	6605, 132132	6603, 211368	603, 31368	303, 12168
ll2	1D	6443, 126656	6443, 213128	6443, 213128	6443, 213128
ll3	1D	3, 96	3, 168	3, 168	3, 168
ll6	1D, 2D	10533, 213036	13223, 430408	13223,430408	13223, 430408
ll7	1D	4807, 86568	7503, 225168	953, 18968	303, 8568
ll9	2D	5883, 117136	2403, 76968	603, 28968	603, 28968
ll12	1D	9005, 180132	3003, 96168	1503, 24168	3, 168
ll21	2D	471, 8520	14403, 460968	123, 58728	123, 58728
ll24	1D	29703, 594096	29703, 950568	29703, 950568	29703, 950568

Most of the programs that access arrays benefit greatly from the implementation of block messages. The greatest improvements occur for loops 1 and 21. Loops 2 and 24 are sequential and the current implementation only generates block messages for references accessed in parallel loops. Loop 6 contains subscript expressions that use non loop variables.

Multiple alignment reduces the number of messages for the programs with producer consumer relations of one-dimensional arrays, such as loops 1,7 and 12.

The volume of communication does not always decrease and varies with program characteristics. In loop 24, where the number of messages exchanged remains the same for all the compiler phases, the volume of communication increases. This is because the implementation of rectangular arrays increases the size of messages required to access array elements in order to accommodate the multiple indices of rectangular arrays.

5.6 Further Work

D-OSC is a prototype implementation that helps us to quantify compiler optimizations for distributed-memory machines. The following are some of the tasks that must be performed to improve D-OSC. A *more efficient run-time system* is needed. There are situations where *run-time reference counting* is necessary. If one processor owns a reference count, each remote processor that updates the reference counter must contact this processor. When deallocating an array, the responsible processor must notify all processors that have partial copies of the array to deallocate the space. The implementation of *function call parallelism* is very easy under the activation-based model. However, inter-functional analysis is required to determine when and where to spawn functions. Currently loops are always distributed over all processors. If an analysis phase can *estimate the computation cost of a loop body*, then it is possible to generate code that decides the number of processors to be used. *Parallel I/O* must be implemented.

6 Architecture Support for Multithreaded Execution

Multithreaded execution has been proposed as a model for parallel program execution. As a model, or rather a family of models, multithreading views a program as a collection of concurrently executing sequential threads that are asynchronously scheduled based on the availability of data. This definition is intentionally wide in that it attempts to capture the common features among various multithreaded execution models proposed to date. It is important to note that in this definition the multithreaded execution model does not specify any form of memory hierarchy (it is common though to expect a single logical address space, shared by many threads and mapped over several nodes), any specific language feature, whether threads are user specified or compiler generated, the mechanism for communication and/or synchronization among threads, or the order of thread execution. There is no standard definition of a thread. In this document we will define a thread as the set of sequential instructions executed between two synchronization points. Note that this definition does not preclude any architecture from exploiting the instruction level parallelism within a thread or the locality of access to a storage hierarchy.

Because of its functional properties, the Sisal language is particularly well suited as a source for multithreaded code. In this Section we present some results related to the evaluation of multithreaded execution. The performance of multithreaded execution is determined by the complex interaction of a number of inter-related architectural and compilation issues such as code generation, thread firing rules, synchronization schemes and thread scheduling. The relation between these issues and the tradeoffs between various alternatives for each of these issues is complex and requires extensive experimental evaluation. For example, the thread firing rule (which determines when threads are enabled) can be based on either a *blocking* or a *non-blocking* strategy. The blocking strategy is adopted in Iannucci's Hybrid Architecture [20], the Tera MTA [2] and the EARTH machine [19]. The non-blocking strategy is adopted in Monsoon [28, 29], *T [25] and the EM-4 [34] among others. The Threaded Abstract Machine (TAM) [13] is a software implemented multithreaded execution that has been ported to a number of platforms (such as the TMC CM-5 and the Cray T3D), it implements the non-blocking model.

In this section we summarize the results of an experimental and quantitative evaluation of these two execution models. The evaluation includes their respective code generation strategies, its implications on data distribution and access and the performance of their respective storage hierarchies.

6.1 Blocking and Non-Blocking Models

The two multithreaded execution models considered here are based on data-driven dynamic execution with statically generated threads. This section presents a detailed description of these two models.

The blocking thread execution model: In this model a thread may be suspended and its execution resumed later. This model requires the underlying architecture to support context switching: i.e., the saving of the thread state and the selection of a new thread. Usually, a thread is suspended after initiating a long latency operation such as a remote memory access.

In this model the synchronization and storage mechanisms rely on the Frame model: A frame represents a storage segment associated with each *invocation* of a code-block[5]. The Frame model is used in several multithreaded machines (e.g. TAM [13], StarT-NG [3] and the EM-4 and EM-X [21]). All the threads within the code-block instance refer to its associated frame to store and load data values. Frames are of variable size and contiguously allocated in the virtual address space. The size of a frame is determined by the maximum number of data values associated with the code-block. When an instance of a particular code-block is invoked, a frame is first allocated in local memory of a processor and all the data values generated within that code-block instance will be stored in that frame. The virtual address carried by a token is of the form:

<*frame pointer, frame offset*>

A synchronization slot in the frame is associated with each thread. The synchronization slot is a counter initialized with the count of the number of the inputs to the thread and is decremented with the arrival of each input. The thread is ready when the count reaches zero. A data value that is shared (i.e. read) by several threads in the same frame occupies only one location. Data values generated by the executing threads are sent to the Synchronization Unit which writes them in the frame. The frame is deallocated when all the threads in the code-block have terminated.

The non-blocking thread execution model: In this model a thread is activated only when all its input parameters are available. Therefore, once a thread starts its execution it runs until termination. All memory accesses are performed as split-phase accesses: the request is issued by a thread but the result is returned to another thread. In this mode the thread never has to block, and be switched out, while waiting for a remote memory access.

The synchronization and storage mechanisms for the non-blocking threads is the Framelet model. A framelet is a fixed sized unit of storage that is associated with each thread instance. Each framelet has one synchronization slot for that thread instance. In the Framelet model a data value that is shared among several threads within a same code-block would be replicated in the framelet of each thread instance. The framelet is deallocated when the thread instance completes its execution. Because their size is fixed, framelets are aligned with cache blocks. The virtual address of a data value in the Framelet model is of the form:

<*context #, thread #, framelet offset*>

Example. A code-block consisting of four threads is shown in Figure 1. The corresponding Frame memory model is shown in the Figure 2. The input *a*

[5] A code-block is a semantically distinguishable unit of code such as a loop or function body.

which is used by both threads A and B is stored at only one place in the frame memory. Each of the values in the frame memory is accessed by the frame base address and the offset into the frame. The first four slots are the counters for the threads. Thus when value c is stored only the counter for D is decremented. But when a is stored both counter for A and B are decremented but only one copy of a is stored in the frame.

The Framelet memory model corresponding to the same code block is shown in Figure 3. There are four separate framelets. Each framelet contains the counter for the corresponding thread and a memory location for all the inputs to the thread. Hence framelet A corresponds to one particular activation of thread A. The a is stored in the framelets of both threads A and B and both counters are decremented. This accomplished as two separate store operations.

6.2 Code Generation

The source language used for the generation of multithreaded code is Sisal. The compilation process converts the programs into two intermediate forms: MIDC-2 (non-blocking) and MIDC-3 (blocking) which are both derived from the *Machine Independent Dataflow Code (MIDC)* [33]. MIDC is a graph structured intermediate format: The nodes of the graph correspond to the von Neumann sequence of instructions and the edges represent the transfer of data between the nodes. MIDC has been used to generate the executable code for other multithreaded machines (e.g Monsoon and EM-4). Both MIDC-2 and MIDC-3 are highly optimized codes with optimization done both at the inter- and intra-thread level.

The code generation compiler is guided by the following objectives [32] :

– Minimize synchronization overhead: by merging threads (thread fusion) and by allocating related threads to the same code block (in the blocking model).
– Maximize intra-thread locality: achieved by thread fusion.
– Assure deadlock-free threads: circular dependencies can create a potential for deadlock.
– Preserve functional and loop parallelism in programs.

The first phase of the code generation is the same for both models, it involves compiling the Sisal programs to IF2 using *OSC* [10].

The second phase differs for the two models in the handling of structure store accesses and the data storage models (frames or framelets). The long latency operations consist of remote memory reads, memory allocations, function calls and remote synchronizations. The remote memory references can be handled either as a *split-phase* access or a *single-phase* access. In the split-phase access the request is sent by one thread and the result is forwarded to another thread. In the single-phase access the result is returned to the same requesting thread. In the non-blocking model all remote accesses are split-phase. The blocking model uses both types of accesses: the code is analyzed at compile time to identify remote and local accesses. Remote accesses are implemented by split-phase operations while local accesses are regular memory access.

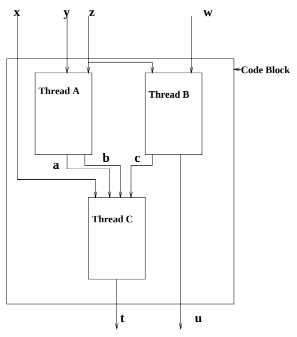

Fig. 1. Code block with Three threads.

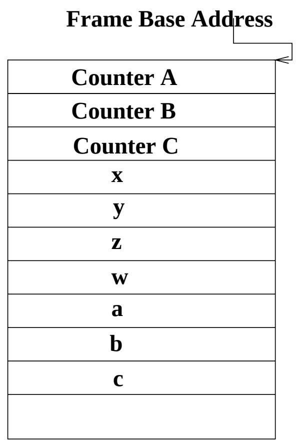

Fig. 2. Frame Memory Representation.

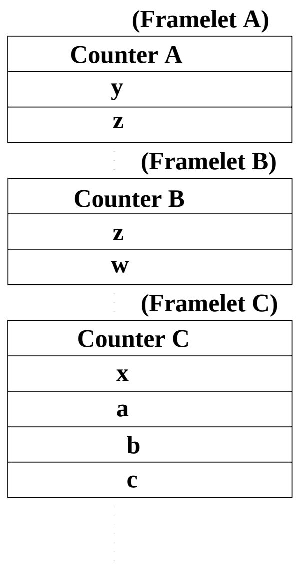

Fig. 3. Framelet Memory Representation.

– In the non-blocking model (MIDC-2 form) all structure store accesses are turned into split-phase accesses. A split-phase access terminates a thread: the request is sent by a thread but the result is returned to another thread. In this model a thread has never to block on a remote memory access. This model does not make any assumption regarding data structure distribution.
– In the blocking model (MIDC-3) the IF2 graph is statically analyzed to differentiate between local and remote structure store accesses: a local access does not terminate a thread while a remote one does. If the result of a structure store access is used within the same code-block where the access request is generated, the access is considered local. In this case, the thread will block until the request is satisfied. This model relies on a static data distribution to enhance the locality of access. Note that a data structure is often generated in one code block and used in several others in which case only one of the consumer code-blocks would have a local access.

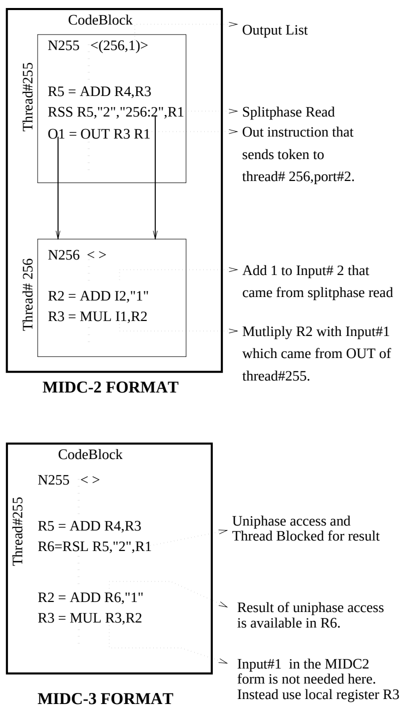
CodeBlock
N255 <(256,1)>
Thread#255
R5 = ADD R4,R3
RSS R5,"2","256:2",R1
O1 = OUT R3 R1
Output List
Splitphase Read
Out instruction that sends token to thread# 256,port#2.
N256 < >
Thread# 256
R2 = ADD I2,"1"
R3 = MUL I1,R2
Add 1 to Input# 2 that came from splitphase read
Mutliply R2 with Input#1 which came from OUT of thread#255.
MIDC-2 FORMAT
CodeBlock
N255 < >
Thread#255
R5 = ADD R4,R3
R6=RSL R5,"2",R1
R2 = ADD R6,"1"
R3 = MUL R3,R2
Uniphase access and Thread Blocked for result
Result of uniphase access is available in R6.
Input#1 in the MIDC2 form is not needed here. Instead use local register R3
MIDC-3 FORMAT

Fig. 4. MIDC-2 and MIDC-3 code examples.

Example The example in Figure 4 demonstrates the difference between MIDC-2 and MIDC-3. In MIDC-2, *Thread* 255 performs a structure memory read operation. The read is performed as a split-phase access where the result is sent to *Thread* 256. *Thread* 255 does not block, it continues execution until termination. When the results of the split-phase read is available it is forwarded to *Thread* 256 which starts execution when all its input data is available. There are no restriction on the processor on which *Thread* 255 and *Thread* 256 are executed.

In the MIDC-3 code, *Thread* 255 and 256 belong to the same code-block. The read structure memory operation is a local single phase operation. Hence, the two threads become a single thread. The thread blocks when the read operation is encountered and waits for the read request to be satisfied.

Discussion of the Models. The main differences between the blocking and non-blocking models lie in their synchronization and thread switching strategies. The blocking model requires a complex architectural support to efficiently switch between ready threads. The frame space is deallocated only when all the thread instances associated with its code block have terminated execution which is determined by extensive static program analysis. The model also relies on static analysis to distribute the shared data structures and therefore reduce the overhead of split-phase accesses by making some data structure accesses local. The non-blocking model relies on a simple scheduling mechanism: data-driven data availability. Once a thread completes execution, its framelet is deallocated and the space is reclaimed.

The main difference between the Frame model and the Framelet models of synchronization is the token duplication. The Framelet model does require that variables which are *shared* by several threads *within* a code block be replicated to all these threads while in the Frame model these variables are allocated only once in the frame. The advantage of the Framelet model is that it is possible to design special storage schemes [31] that can take advantage of the locality of the inter-thread and intra-thread locality and achieve a cache miss rate close to 1%.

6.3 Summary of Performance Results

This section summarizes the results of an experimental evaluation of the two execution models and their associated storage models. A preliminary version of these results was reported in [4], detailed results are reported in [5].

The evaluation of the program execution characteristics of these two models shows that the blocking model has a significant reduction in threads, instructions, and synchronization operations executed with respect to the non blocking model. It also has a larger average thread size (by 26% on average) and, therefore, a lower number of synchronization operations per instruction executed (17% lower on average).

However, the total number of accesses to the Frame storage, in the non-blocking model, is comparable to the number of accesses to the Framelet storage in the blocking model. Although the Frame storage model eliminates the replication of data values, the synchronization mechanism requires that two or more synchronization slots (counters) be accessed for each shared data. The number of synchronization accesses to the frames nearly offsets all the redundant accesses. In fact the size of the trace of accesses to the frames is less than 3% smaller than the framelet trace size. Hence, synchronization overhead is the same for the frame and framelet models of synchronization.

The evaluation also looked at the performance of a cache memory for the Frame and Framelet models. Both models exhibit a large degree of spatial locality in their accesses: In both cases the optimal cache block size was 256 bytes. However, the Framelet model has a much higher degree of temporal locality resulting in an average miss rate of 1.82% as opposed to 5.29% for the Frame model (both caches being 16KB, 4-way set associative with 256 byte blocks).

The execution time of the blocking model is highly dependent on the success rate of the static data distribution. The execution times for success rates of 100% or 90% are comparable and outperform those of the non blocking model. For a success rate of 50%, however, the execution time may be higher than that of the non blocking model. The performance, however, depends largely on the network latency. When the network latency is low and the processor utilization high, the non blocking model performs as well as the blocking model with a 100% or 90% success rate.

7 Conclusions and Future Research

The functional model of computation is one attempt at providing an implicitly parallel programming paradigm[6]. Because of the lack of state and its functionality, it allows the compiler to extract all available parallelism, fine and coarse grain, regular and irregular, and generate a partial evaluation order of the program. In its pure form (*e.g.,* pure Lisp, Sisal, Haskell), this model is unable to express algorithms that rely explicitly on state. However, extensions to these languages have been proposed to allow a limited amount of stateful computations when needed. Instead, we are investigating the feasibility of the declarative programming style, both in terms of its expressibility and its run-time performance, over a wide range of numerical and non-numerical problems and algorithms, and executing on both conventional and novel parallel architectures. We are also evaluating the ability of these languages to aid compiler analysis to disambiguate and parallelize data structure accesses.

On the implementation side, we have demonstrated how multithreaded implementations combine the strengths of both the von Neumann (in its exploitation of program and data locality) and of the data-driven model (in its ability to hide latency and support efficient synchronization). New architectures such as TERA [2] and *T [26] are being built with hardware support for multithreading. In addition, software multithreading models such as TAM [13] and MIDC [8]), are being investigated.

We are currently further investigating the performance of both software-supported and hardware-supported multithreaded models on a wide range of

[6] Other attempts include the vector, data parallel and object-oriented paradigms.

parallel machines. We have designed and evaluated low-level machine independent optimization and code generation for multithreaded execution. The target hardware platforms will be stock machines, such as single superscalar processors, shared memory, and multithreaded machines. We will also target more experimental dataflow machines, (*e.g.*, Monsoon [18, 37]).

References

1. S. J. Allan and R. R. Oldehoeft. Parallelism in SISAL: Exploiting the HEP architecture. In *19th Hawaii International Conference on System Sciences*, pages 538–548, January 1986.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. P. ortfield, and B. Smith. The Tera computer system. In *Proceedings 1990 Int. Conf. on Supercomputing*, pages 1–6. ACM Press, June 1990.
3. B. S. Ang, Arvind, and D. Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. Technical Report 354, LCS, Massachusetts Institute of Technology, August 1994.
4. M. Annavaram and W. Najjar. Comparison of two storage models in data-driven multithreaded architectures. In *Proceedings of Symp. on Parallel and Distributed Processing*, October 1996.
5. M. Annavaram, W. Najjar, and L. Roh. Experimental evaluation of blocking and non-blocki ng multithreaded code execution. Technical Report 97-108, Colorado State University, Department of Computer Science, www.cs.colostate.edu/ ftppub/TechReports/, March 1997.
6. J. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21(8):613–641, 1978.
7. A. Böhm and J. Sargeant. Efficient dataflow code generation for sisal. Technical report, University of Manchester, 1985.
8. A. P. W. Böhm, W. A. Najjar, B. Shankar, and L. Roh. An evaluation of coarse-grain dataflow code generation strategies. In *Working Conference on Massively Parallel Programming Models*, Berlin, Germany, Sept. 1993.
9. D. Cann. *Compilation Techniques for High Performance Applicative Computation*. PhD thesis, Colorado State University, 1989.
10. D. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.
11. D. Cann. Retire FORTRAN? a debate rekindled. *CACM*, 35(8):pp. 81–89, Aug 1992.
12. D. Cann. Retire Fortran? a debate rekindled. *Communications of the ACM*, 35(8):81–89, 1992.
13. D. E. Culler et al. Fine grain parallelism with minimal hardware support: A compiler-controlled Threaded Abstract Machine. In *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991.
14. J. T. Feo, P. J. Miller, and S. K. Skedzielewski. Sisal90. In *Proceedings of High Performance Functional Computing*, April 1995.

15. M. Forum. *MPI: A Message-Passing Interface Standard*, 1994.

16. G. Fox, S. Hiranandani, K. Kennedy, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251-1892, 1990.

17. D. Garza-Salazar and W. Böhm. Reducing communication by honoring multiple alignments. In *Proceedings of the 9th ACM International Conference on Supercomputing (ICS'95)*, pages 87–96, Barcelona, 1995.

18. J. Hicks, D. Chiou, B. Ang, and Arvind. Performance studies of Id on the Monsoon dataflow system. *Journal of Parallel and Distributed Computing*, 3(18):273–300, July 1993.

19. H. Hum, O. Macquelin, K. Theobald, X. Tian, G. Gao, P. Cupryk, N. Elmassri, L. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A design study of the EARTH multiprocessor. In *Parallel Architectures and Compilation Techniques*, 1995.

20. R. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report 418, Ph. D Dissertation Technical Report TR-418, Laboratory for Computer Science, MIT, Cambridge, MA, June 1988.

21. Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the* 22^{th} *Annual International Symposium on Computer Architecture*, pages 14–23, June 1995.

22. J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL-Streams and Iterations in a Single Assignment Language, Language Reference Manual, version 1. 2. Technical Report TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.

23. P. Miller. TWINE: A portable, extensible sisal execution kernel. In J. Feo, editor, *Proceedings of Sisal '93*. Lawrence Livermore National Laboratory, October 1993.

24. P. Miller. Simple sisal interpreter, 1995. ftp://ftp.sisal.com/pub/LLNL/SSI.

25. R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the* 19^{th} *Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

26. R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

27. R. Oldehoeft and D. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, January 1988.

28. G. Papadopoulos. Implementation of a general-purpose dataflow multiprocessor. Technical report TR-432, MIT Laboratory for Computer Science, August 1988.

29. G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the* 17^{th} *Annual International Symposium on Computer Architecture*, pages 82–91, June 1990.

30. J. Rannelletti. *Graph Transformation algorithms for array memory optimization in applicative languages.* PhD thesis, U. California, Davis, 1987.
31. L. Roh and W. Najjar. Design of storage hierarchy in multithreaded architectures. In *IEEE Micro*, pages 271–278, November 1995.
32. L. Roh, W. Najjar, B. Shankar, and A. P. W. B. öhm. Generation, optimization and evaluation of multith readed code. *J. of Parallel and Distributed Computing*, 32(2):188–204, February 1996.
33. L. Roh, W. A. Najjar, B. Shankar, and A. P. W. Böhm. An evaluation of optimized threaded code generation. In *Parallel Architectures and Compilation Techniques*, Montreal, Canada, 1994.
34. S. Sakai, K. Hiraki, Y. Yamaguchi, and T. Yuba. Optimal Architecture Design of a Data-flow Computer. In *Japanese Symposium on Parallel Processing*, 1989. in Japanese.
35. S. Skedzielewski and R. Simpson. A simple method to remove reference counting in applicative programs. In *Proceedings of CONPAR 88*, Sept 1988.
36. S. K. Skedzielewski, R. K. Yates, and R. R. Oldehoeft. DI: An interactive debugging interpreter for applicative languages. In *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, pages 102–109, June 1987.
37. K. Traub. Monsoon: Dataflow Architectures Demystified. In *Proc. Imacs 91 13*th *Congress on Computation and Applied Mathematics*, 1991.
38. M. Welcome, S. Skedzielewski, R. Yates, and J. Ranelleti. IF2: An applicative language intermediate form with explicit memory management. Technical Report TR M-195, University of California - Lawrence Livermore Laboratory, December 1986.

This article was processed using the LaTeX macro package with LLNCS style