Control Flow Extensions To The Dataflow Language SISAL

Vasu Vijayaraghavan
Krishna M. Kavi
and
Behrooz Shirazi

The University of Texas at Arlington

Abstract

The proponents of the dataflow programming model have been citing its advantages: exposing fine grain parallelism, freedom from side-effects, and synchronization based purely on data dependencies. Opponents of the model cite large overheads incurred in utilizing the fine grain parallelism, the lack of facilities to specify coarse grain parallelism, and lack of explicit control of the concurrent activities. More recently, there have been numerous proposals for combining some control flow concepts with the dataflow model. Our research differs from these primarily because we attempt to bring the dataflow language closer to more popular control flow based concurrent programming languages, while retaining the fundamental properties of dataflow model. Our main goal is to provide the programmer with mechanisms to define and control coarse grain concurrency. To this end, we have extended the SISAL language (for Sequent) by including fork, join and break constructs.

Index Terms: Dataflow, Control flow, Concurrent Processes, Fork, Join, Dataflow languages, SISAL, Shared Memory Multiprocessors

1. Introduction

In this paper we describe our attempt to bring the dataflow language SISAL closer to more popular control flow based concurrent programming languages, while retaining the fundamental properties of the dataflow model. Our main goal is to provide the programmer with mechanisms to define and control coarse grain concurrency. In addition to user specified (coarse grain) concurrency, the SISAL compiler can still extract parallelism based on iterative constructs within these processes.

The dataflow model of computation has attracted considerable interest during the seventies and the eighties, primarily because of its data-driven mode of execution, freedom from side-effects, ability to exploit fine-grain parallelism, and synchronization based purely on data dependencies. This period of research led to new programming languages ([Ackermann 82], [Arvind 78], [Ashcroft 77], [McGraw 82, 85]), proposals for dataflow machines ([Arvind 80, 82, 90], [Gurd 85], [Papadopoulos 88]), and concurrency models ([Brock 83], [Deshpande 89], [Staples 85]).

However, the fine grain parallelism and the value-oriented execution (no assignments or only single-assignments) implied by dataflow have made the model inefficient to implement on von Neumann machines. The single-assignment (or no assignment) and the freedom from side-effects are radically different from concepts of conventional control-flow based programming languages, making dataflow unpopular with programmers.

The gap between dataflow and the von Neumann model has generated a renewed interest in combining the two models. Examples include, compiling dataflow languages on conventional architectures [Cann 90], variable and coarse grain dataflow ([Bic 87], [Gaudiot 84]), examining hybrid architectures comprising von Neumann and dataflow features ([Arvind 89], [Boehrer 87], [Janucci 88]), and compiling conventional languages on to dataflow machines [Beck 90]. This paper continues the research trend and explores the possibility of combining the advantages of the dataflow model with some control-flow based concurrent programming language constructs.

It is our belief that dataflow affords very little for the programmer in defining, controlling and synchronizing concurrent computations. Most dataflow compilers (written for conventional computers) partition the program into independently executable blocks of code relying on iterative constructs and function definitions. Scheduling, controlling and synchronization of processes are the responsibilities of the compiler. As a contrast, control-flow based concurrent programming languages provide a fairly wide range of constructs for the user to specify parallelism. However, these languages do not expose fine grain parallelism. Even though compiler based approaches can detect fine grain concurrency, they are used primarily for scheduling independent (coarse grain) tasks.
Ianucci [Ianucci 88] also examines a hybrid architecture. Here, the dataflow features essential for minimizing memory latency and synchronization overheads are encapsulated in a von Neumann machine using a "parallel machine language" which is grafted on top of a traditional machine base. The availability of referenced memory (I-structures) is determined using presence bits. A context switch takes place when a process (thread of execution) is blocked from a reference to an unavailable memory. The preliminary results presented indicate a case for explicit synchronization instructions.

As a next step in building hybrid architectures, Nikhil and Arvind [Nikhil 89] (also the Monsoon processor [Papadopolous 88]) propose a RISC-like machine architecture. In this approach, each dataflow node is identified by an instruction pointer (Ip) - frame pointer (Fp) tag. The frames refer to independent threads of execution (function activations). Using the Fp-Ip tag, computations from different threads can be interleaved on a single pipelined RISC-like processor. The successor instruction (or computation) within a thread is determined by the current Fp-Ip tag (either the Ip is incremented, or a new Ip based on branch instructions is computed). Dependency (communication and synchronization) among concurrent threads can be described using machine level fork and join instructions. In a related research, Beck and Pingali [Beck 90] show that imperative language programs can be translated into dataflow graphs and executed on a dataflow machine. This attempts to exploit both fine grain as well as coarse grain parallelism in imperative language programs. This group of researchers believe that the dataflow model of computation can subsume von Neumann model.

On a different end, researchers have been working on compilers to execute dataflow programs on conventional multiprocessors. Sarkar and Cann [Sarkar 90] have extended the SISAL compiler to include automatic determination of parallelism based on control and data dependencies, node execution times and multiprocessor overhead parameters, over and above the parallelism specified by the foralls, function calls and loops that produce/consume streams. The partitioning carried out is based on execution profiling and execution time estimations. Several other partitioning and mapping algorithms for creating coarse grain dataflow threads have been proposed during the past decade.

Bic [Bic 87] proposes sequential code segments (SCS's) to permit coarse grain parallelism. However, the definition of SCS's is not under user control, but automatically generated by dataflow compilers. The idea of SCS's is similar to Gaudiot's [Gaudiot 84] macro dataflow nodes. Wu and Thorelli [Wu 90] suggest an Extended Dataflow Model (EDA) which is a hybrid of the classical von Neumann machine and the traditional dataflow model. An address space is added to a dataflow graph that is executed in a data-driven manner. Token passing at the task level and variable access at the instruction level provide means for communication and synchronization.

Ruighaver and Yeo [Ruighaver 90] argue strongly in favor of explicit parallelism constructs and the use of dataflow concepts. However they deem the process model error-prone - even the most elaborate synchronization primitives are not immune to deadlock and program testing is complicated by the inability to create and recreate the conditions for which the program fails. They extend a subset of Modula2 to Parallel Modula2 (PM2), by providing explicit parallelism constructs and object based abstraction.

Our research differs from these primarily because we attempt to bring the dataflow language closer to more popular control flow based concurrent programming languages, while retaining the fundamental properties of the dataflow model. Our main goal is to provide the programmer with mechanisms to define and control coarse grain concurrency. Because of the use of the dataflow model, fine grain parallelism within user specified processes still exists and can be exploited when SISAL programs are executed on dataflow processors. Moreover, a compiler can ignore user specified coarse grain parallelism in favor of optimizing across process boundaries.

2. Control Flow Extensions To SISAL
2.1 SISAL Language Overview
SISAL is a single assignment programming language with value oriented semantics. Single assignment means that each variable has at most one value assigned to it during program execution. Such languages have no storage related anti or output data dependencies, and yield more parallelism than programming languages with memory update operations and other side effects.

An executable SISAL entity comprises one or more separately compilable units, which include a program, interfaces and modules. The program consists of a set of declarations that define entities imported from other modules, some types and some functions. Any function in a program may be the starting point for execution. A module is syntactically similar to a program, but is not a starting point for execution. It pairs with an interface of the same name to export some of its type and function names. A stand-alone interface declares the relationship between SISAL and a set of subsidiary code written in other languages.

Data types include scalars (boolean, character, integer, real and double), structured types (records & unions, arrays and streams) and functions. Facility for parametrized types exists.
A function is declared by listing its name, the names and types of its formal parameters and types of its result values. The content of a function is one or more expressions whose types correspond to the result types. Reduction is a special form of a function that is applied to collapse a set of values into a single value.

Conventional infix operators combine scalar arithmetic values. Name scoping is done using a 'let' construct. Comprehensive facilities for defining and manipulating array values are provided. Streams (which allow for pipelined parallelism) are produced and consumed using the 'for' expression.

Control by selection is afforded by the 'if' and 'case' constructs. The 'for' construct has two forms for potential parallel as well as sequential evaluation. These forms are analogous to do-all and do-across of conventional concurrent languages.

SISAL programs are converted into data dependence graphs (called IF1 - Intermediate Form I) which exposes fine grain parallelism. Based on data dependencies, execution times and communication costs, the dataflow graphs are grouped to form coarse grain execution threads, and then mapped to run on separate processors.

A small SISAL example program is shown below. The corresponding IF1 is shown in Fig. 1.

```plaintext
define Fun
  function Fun(x, y : integer returns integer, integer)
    for initial
      i := x
      while i <= y repeat
        i = old i + 1;
        returns value of sum i
    end for
    x*y-5
  end function
```

This program executes two (independent) tasks - calculating the sum of consecutive integers between two specified integers (x & y) and the value of (x*y-5). The IF1 graph consists of a compound node (LoopB) for the loop and a multiplication & subtraction node for the other independent computation (see [Cann 90] for a more details on SISAL).

2.2. Concurrency in Dataflow and Control Flow

The reader is referred to [Almasi 89] and [Andrews 83] for a good overview of programming language constructs to support concurrent programming. Three issues underlie concurrent programming -1) Expressing concurrent execution, 2) Communication and Synchronization, and 3) Debugging.

Expressing Concurrent Execution. Control flow languages specify concurrent execution by using coroutines, fork & join statements, cobegin & coend statements and process declarations. In its most general form, the fork creates additional processes while join ensures that, of all the processes forked off, only one (determined non-deterministically based on who completes last) proceeds execution beyond the join statement; all the other processes quit. Process declarations consist of specifying named blocks of code for execution in parallel. Multiple activations of a particular code block is possible. The statements within a process (code block) are executed sequentially.

In dataflow languages, concurrent execution is implied by the independence of computations (actors or nodes in dataflow graphs). All such independent computations can be executed in parallel (based on the availability of processing resources). Otherwise, a computation is enabled only when all its necessary values (inputs) are made available by some preceding computations. When executing on dataflow machines, each computation (dataflow node) is treated as an independent thread (hence excessive overheads). When programs written in dataflow languages such as SISAL are compiled on conventional multiprocessing systems, the programs are partitioned and grouped to form coarse grain processes. Such partitioning and mapping algorithms strive to achieve maximal parallelism at minimal overhead costs [Sarkar 86, 90].

Communication and Synchronization. Communication and Synchronization among processes is required to ensure sequence and data control. Control flow languages achieve communication using either shared variables or messages. Messages can be exchanged either synchronously or asynchronously. Communication based on shared variables requires synchronization using semaphores, spinlocks, or monitors. Flow control communication (and synchronization) is achieved using synchronization points or barriers.

Communication among dataflow computations is needed only to exchange data (usually implying message-passing mode). Synchronization among concurrent computations is implied purely by the data dependencies. Some dataflow models support other forms synchronization while accessing data structures (e.g. I-structures). When dataflow programs are compiled on conventional shared memory multiprocessors, communication and synchronization among concurrent (coarse grain) processes is achieved using shared variables.

Debugging. Debugging of concurrent programs is complicated by the indeterminacy in program behavior over different executions, lack of mechanisms for selecting a priori break points, and the difficulties in tracing program flow [Ying 90]. Most concurrent control-flow language debuggers operate on a uniprocessor in a multitasking environment. Though
break points may be specified, the indeterminacy of program behavior over different executions still exists. In dataflow, break points may be simulated by denying a computation (or process) the data inputs required to proceed. In supporting break points and tracing facilities with dataflow languages one must understand the implications of the following properties.

1. Freedom from side-effects. Dataflow variables can be assigned values at most once, and thus there are no (modifiable) shared global variables. Some dataflow languages permit the definition of scopes with variables using constructs like let. Due to the lack of shared variables, the implementation of break points requires the collection and modification of the values from each process in a concurrent environment.

2. Asynchronous Execution. Computations execute as soon as all the inputs are available. A computation cannot be scheduled deterministically. Since synchronization is based purely on data dependencies, control flow synchronizations appear artificial. The implication here is that artificial boundaries for synchronization must be introduced in dataflow model.

3. Memory Latency. When using I-structures, the delays involving in accessing memory can be viewed as synchronization delays. I-structure access is usually implemented as a split transaction. However, to implement fork, join and breakpoints, we must view the memory reference as an atomic action.

2.3. Extended SISAL

2.3.1. Process Declaration.

Use of Fork and Join constructs mandates the incorporation of process declarations to identify sections of code that comprise coarse grain threads of execution. In our extended SISAL, process declarations may appear anywhere in a program and can include any SISAL statements, functions and modules. They are written as:

```sisal
process process-name
    SISAL statement ;
    SISAL statement ;
    ....
    ....
function F1(....)
    module module-id
end process
```

Processes are invoked either by an explicit call or by using forks, stopped using a break, restarted on user input, and synchronized and killed using a join.

2.3.2. Fork and Join

Fork and join can specify parallel execution at the highest level and are perhaps the most natural way of expressing coarse grain parallelism. By providing fork and join constructs in SISAL, an opportunity is provided to the programmer to specify units of the program that can be executed concurrently, thus generating coarse grain parallelism which can be directly executed on contemporary architectures. It is clear that it is the programmer who possesses a better knowledge and understanding of the problem domain, and will be able to assist the compiler in identifying independent program partitions using these constructs. SISAL nodes execute asynchronously. Fork and join constructs allow the programmer to control the execution of independent processes and force synchronization at process boundaries. Normally, statements within a process (code block) are executed sequentially. However, this does not preclude the exploitation of additional parallelism within these code blocks (relying on iterative constructs).

The fork statement looks like:

```
fork (P1, P2, ..., Pn);
```

where P1 is a process name. In fork, processes can be declared using P1: Module_Name or P1: Function_Name (in addition to using previously defined processes). The fork statement will cause the n processes to be spawned off and these may run on different processors based on the availability of the resources. It should be noted that the process-names, module-ids and function names are distinct.

The join statement looks like:

```
join (P1, P2, ..., Pn);
```

A join statement is placed in each process Pi\(\in\)\{P1, P2, ..., Pm\} that participate in the flow synchronization. The join ensures that the processes executing the join will stop to synchronize by waiting for the other processes. In the present implementation, a separate dataflow node is created for coordinating the "joining" processes. A IFI graph showing fork and join nodes is shown in Fig. 2.

The compiler interprets the fork statement and marks each of the process blocks for concurrent execution. At present, we are using SISAL provided runtime scheduling of the threads on available processors. When using a pool of "light weight" processes [Accetta 86], each process Pi spawned by a fork statement can be bound to a process in the pool. Upon reaching the join (after which spawned processes are terminated), processes return to the pool. A join node is inserted at the end of each process block which ensures that the process terminates itself after that node. The output from each of these join nodes forms the input to a reduction node called the end-join node which ensures that all the forked processes have terminated. It
is not necessary for the processes spawned by a specific fork statement to culminate in the same join - subsets of the processes may terminate on different join statements.

Remarks. It should be noted that the inherent fine grain parallelism within each process P_i can still be exploited, notwithstanding the coarse grain parallelism defined by the fork construct. Depending on the size of the processes, each user specified process may be partitioned into a set of concurrent processes (based on the data dependencies, execution costs and communication costs). Also, where appropriate (as a user option), a compiler may ignore process boundaries and optimize across them. None of these will alter the operation or the semantics of the join operation.

Fork and Join do not add any runtime overhead. In fact, by using the concept of "light weight" processes, processes spawned by fork construct can be bound statically to the process code blocks.

Part of a SISAL program showing fork and join usage is as below:

```
process sumconsec
  for initial
    i := x
    while i <= y repeat
      i = old i + 1;
    returns value of sum i
  end for
end process

process multsub
  x*y-5
end process

{parent process statements follow}
```

In this program, two process blocks are declared. The fork instantiates three processes p1, p2, and p3 each executing the sumconsec, multsub and multsub blocks of code possibly concurrently on either the same or different processors. Note the possibility of multiple instances of the same process code.

2.3.3. Creating Break Points.

"Break" stops program execution, and awaits an "external" impetus to continue execution beyond the break point. This permits the freezing of the state of execution of dataflow processes at chosen points of execution. The values of variables can be examined, and these values can be modified before stopped processes proceed. This is the primary goal of our break points.

A break construct looks like

```
break break_name (arguments);
```

Break points can be named using the break_name such that only the processes using the same break_name participate in the debugging activity at that break point. Arguments specify the values that can be examined, and modified. Each process participating in a given break point must include the break construct. A process halts when it executes the break statement. It can resume execution beyond the break point only under an external signal. A graph showing the break node is shown in Fig.3.

As can be seen from the IF1 graph, the compiler creates a coordinating process for each named break point. The debugging actions to be performed at any break point is specified by the user (the break handling subgraph).

Part of a SISAL program showing break & restart usage is given below:

```
process A
  for initial
    i := x
    break breakone(i, x returns i);
    while i <= y repeat
      i = old i + 1;
    returns value of sum i
  end for
end process

process B
  for initial
    i := y
    break breakone(i, x returns i);
    while i <= x repeat
      i = old i + 1;
    returns value of sum i
  end for
end process

At breakpoint breakone, process A breaks just after initializing the value of i and process B breaks after execution of the loop. At this time, the coordinating process displays the values of i & x from process A and the values of x & y from process B. The user may modify the value of i in process A and the value of y in process B. These modified values are returned to the breakpoint's immediate successor in each process as input.
When the process is re-enabled through an external input, process A starts executing the loop, while process B computes \((x*y-5)\) and completes execution. Process A has another break statement, breaktwo immediately after the loop. At breaktwo, the values of \(i\), \(x\), and \(y\) in process A are displayed.

Remarks: Debugging a parallel program is a difficult task complicated by the fact that it is not trivial to stop computation at all processors at the same point of execution repeatedly even though this is essential for state comparison in debugging. This is because of communication delays and the fact that multitasking and multiprogramming may cause scheduling variance. The break construct detailed in this paper ensures that the break takes place at well defined boundaries of computation (easily defined in a dataflow program), so that repeated execution of the break will cause the computation to stop at the same point across all processors. The break coordinating processes can invoke user supplied break-handlers (dataflow functions) to facilitate interactive debugging. We have not shown the break-handler in the above example.

2.4. Implementation Considerations

2.4.1. Language issues.

Process declarations are implemented by having the compiler identify those blocks of code as independently executable. The IF1 graph is simply annotated by adding labels to the start and end nodes of the code block.

No separate fork node is required as the graph has already been annotated to indicate the processes which can be forked off. The end-join node in the present implementation consists of a Noop node having \(n\) inputs, from each of the join nodes in each of the processes. Only one output from this node is required to represent the semantics of the join construct.

The break node in each of the processes is a Noop node. The coordinating break node is a compound node which accepts a number of inputs (as specified by the break construct), and these are transmitted to the break-handling subgraph. The break-handling sub-graph performs actions as indicated by the user. The outputs (modified values) of the coordinating break node will be used by the stopped processes. The break-handling subgraph interacts with the user and restarts the stopped tasks only when directed by the user. The simplest form of the break-handling subgraph would be a node representing external assignment of a value to a variable. Note that all these nodes are implemented using presently available SISAL nodes.

2.4.2. Run Time Issues.

The SISAL runtime multitasking system is sufficient to support the fork, join and break constructs. The runtime system supports the parallel execution of SISAL programs, provides general purpose dynamic storage allocation, implements operations on major data structures and interfaces with the operating system for input/output and command line processing [Foo 90].

Support for parallel execution is in the form of threads or lightweight processes just as in the Mach operating system [Accetta 86]. A command line option specifies the number of operating system processes to instantiate for the duration of the program. This is done by the compiler using user specified parallelism and any finer grain parallelism that can be extracted by the compiler (based on iterative constructs). The number of light weight processes remains constant, and these processes form of a pool workers awaiting jobs.

Each process block of code is compiled as an independently executable instruction stream (task). When the fork statement is executed, these tasks are placed on the ready list. The available worker processes will then execute the tasks on the ready list. Upon the termination of a task (after "join"ing) worker processes return to the pool of available workers.

The introduction of coarse grain parallelism will not prevent the compiler from extracting additional concurrency based on iterative constructs. In other words, a coarse grain process may contain several concurrent tasks which can also be placed on the ready list.

When a break statement is executed, the process stops. The corresponding break-coordinating process waits till all the processes for that break point stop. Then it executes the (precompiled) break-handling subgraph task. Upon receipt of an external input, it restarts all the stopped threads to continue through termination. Only then will the worker processes return to the pool of available workers.

As can be seen, very little needs to be added to the SISAL runtime multitasking system to support the additional constructs. However, in the near future we propose to use the microtasking provided by the DYNIX operating system on the Sequent. The emphasis will be on statically instantiating the requisite number of light weight processes, utilizing the microtasking and data partitioning libraries and the dynamic scheduling facilities of DYNIX.

Finally, we feel that that additional IF1 node types are needed to fully exploit the capabilities of fork, join and break. These include: a nondeterministic merge, a selector node and a distributor node as shown in Fig 4.

The Selector node accepts a number of inputs \((A, B, C, \ldots)\) and based on a control input "Control", one of the inputs is passed to the output.

The Distributor node accepts one input \(X\), and based on the control input "Control", this input is passed to one of the outputs.
The Non-deterministic Merge node combines its inputs into a stream and passes them to the output. Presently, the ordering is based on their arrival (First come First served).

3. Conclusions and Future Work

Normally, the SISAL compiler creates coarse grain tasks from dataflow programs using iterative constructs and functions. It is our belief that a knowledgeable user will be a better judge of the coarse grain parallelism and he should be provided with mechanisms for the definition and control of the coarse grain parallelism. To this end we extend SISAL by including fork, join and break constructs. Break is primarily for debugging and tracing of the concurrent processes. The compiler can still extract additional parallelism within user specified (coarse grain) concurrency.

The introduction of process boundaries, fork and join will bring dataflow programming languages closer to conventional control-flow concurrent languages. This in turn makes it easier to translate control-flow based parallel algorithms into dataflow languages.

At present we are using SISAL runtime environment (for Sequent) for the creation and management of concurrent tasks. In the near future we are planning to use the microtasking provided by the DYNIX operating system. Additionally, we are investigating the creation of three new IF1 nodes, non-deterministic merge, selection (one of the inputs are selected and passed to output) and distribution (the input is passed to one of the outputs) to better support the semantics of fork, join and break constructs.

Acknowledgements. This work is supported in part by a grant from the State of Texas Coordinating Board on Higher Education, under the Advanced Technology Program, #3656-057.

4. References


Programming Languages & Systems, Jan 1982
pp 44-82.


Fig. 1a IF1 graph of simple SISAL program

Fig. 1b IF1 graph for Loop B node in Fig 1a

Fig. 2 IF1 graph showing fork and join

Fig. 3a IF1 graph showing Break construct
Fig. 3: Typical Break Coordinating Node

Fig. 4: New IF Nodes