Proc. Int. Conf. Aplication-Specific Array Processors, IEEE Computer Society, Aug. 4-7, 1992.

Programming Systolic Arrays

Richard Hughey

Computer Engineering Board
University of California
Santa Cruz, CA 95064
rph@ce.ucsc.edu

Abstract

This paper presents the New Systolic Language as a general solution to the problem systolic programming. The language provides a simple programming interface for systolic algorithms suitable for different hardware platforms and software simulators. The New Systolic Language hides the details and potential hazards of inter-processor communication, allowing data flow only via abstract systolic data streams. Data flows and systolic cell programs for the co-processor are integrated with host functions, enabling a single file to specify a complete systolic program.

1 Introduction

As massively parallel machines and co-processors become more common, tools and languages are needed to harness their vast power. Systolic algorithms are perhaps the most efficient class of algorithm for massively parallel implementation, using regular data flow through a processor network to harness the inherent parallelism of a problem and to avoid communication bottlenecks in the parallel processor [7]. A systolic algorithm has two major parts: a cell program and a data flow specification. The cell program defines the local operations of each processing element, while the data flow describes the communication network and its use. Additionally, systolic applications must be concerned with host to co-processor communication and data access.

This paper presents a language that eases the process of transforming a systolic algorithm into a functioning systolic program. The New Systolic Language (NSL) hides low-level implementation concerns such as the availability of physical connections, the allocation of registers and delay elements, and the avoidance of data and timing hazards; the programmer deals exclusively with an abstract and versatile systolic stream data type. Data flows are defined in terms of these streams and separate systolic cell programs perform local computation on the streams. These two parts of the systolic program are integrated with host functions, enabling host input and output to be hooked directly to the systolic co-processor.

The New Systolic Language is intended to be an abstract, machine-independent programming language. However, it was developed as a high-level interface to the Brown Systolic

Array (B-SYS), and reflects some of the characteristics of that machine, briefly described in Section 3. The prototype NSL system generates B-SYS assembly code which can either be printed out for inspection or sent to the B-SYS simulator and animator (based on the *xtango* algorithm animation package [21]). Interfaces to a B-SYS system and to a general-purpose massively parallel machine are under development, as are many enhancements to and refinements of the language. The New Systolic Language and its stream model of data flow attempt to overcome the difficulties and adapt the advantages of several existing systolic programming methods.

2 Systolic Programming

Systolic specification paradigms can be loosely divided into two categories: systolic programming languages and systolic design languages. Systolic programming languages provide constructs or subroutine libraries for general-purpose systolic processors or multiprocessors as a means of implementing systolic programs on the machine. On the other hand, systolic design systems aid in mapping an algorithm or recurrence relation to the systolic domain. Using a systolic design system, various interconnection networks and data flows may be explored as a precursor to building or programming a systolic array. These systems are especially useful to the producer of single-purpose systolic arrays who can implement *any* systolic data flow in the custom network. Programmable systolic arrays can, in general, accommodate a large selection of data flows despite hardwired restrictions.

Typically, systolic programming methods for existing machines adopt some of the peculiarities of the machine (the current NSL implementation does not entirely avoid this pitfall). For example, W2 and Hearts provide for the replication of a cell program but still require the programmer to access the communication links within the cell program [13, 20]. W2 cell programs include `send` and `receive` statements for placing data on one or the other of Warp's physical queues. In Hearts, logical data connections are defined in an external table and graph (i.e., not in the textual programming language of the cell program), while data *movement* is the result of the cell program's port or channel input and output instructions, a typical problem of shared asynchronous variables and similar constructs [12]. The Apply system on the iWarp multiprocessor features localized cell programs but, being designed specifically for multiprocessor image processing, does not provide general flow directives [24].

General-purpose parallel programming languages such as Paris or C* on the Connection Machine and Occam on Transputer arrays can also be used for systolic programming, but often require the programmer to be even more concerned with low-level architectural details [16, 23]. One advantage of these languages, however, is that both the systolic program and the host or main program can be expressed in a common language, simplifying the interface between the systolic algorithm and the rest of the computer.

Systolic design languages have many advantages over hardware-specific languages. Instead of building up from a specific machine implementation, they move down from abstract systolic specifications. The two most common specification paradigms are dependency mappings, either as matrix transformations or graphs, and recurrence equations, the basis of functional systolic languages. As the semantics of these two paradigms are so closely related, it is often difficult to categorize a given language.

Engstrom and Capello's SDEF systolic design environment [6] is a dependency mapping system that has greatly influenced the specification requirements of the New Systolic Language. SDEF cleanly separates systolic cell programs from the macroscopic systolic data movement, as can be seen in Figure 1. This feature is particularly attractive on the realization that many algorithms share systolic data flow patterns and many different flow

```
Dimension: 2
Orthohedral Bounds:
  lower  upper
    0    degree  i
    0    n       j
Domains:
  name: C type: float dependence: i(0)  j(-1)
  name: X type: float dependence: i(-1) j(0)
  name: Y type: float dependence: i(-1) j(0)
Function: name: POLY
Embedding:
   1 1
   0 1
```

(a) SDE File

```
POLY(C,X,Y)
{
  Y' = C + X*Y;
}
```

(b) F File

Figure 1: SDEF program for Horner's method.

patterns can solve a specific problem, often with identical cell programs. One drawback of the SDEF system (shared by Hearts' graphical data flow specification) is the multiple views required for each application: cell programs, systolic mappings, and inputs and outputs are not specified in a single, common, language, but each has its own input format.

Because of the close relationship between recurrence equations and systolic algorithms, functional or rule-based languages are one of the most precise ways to specify a systolic algorithm. Chandy and Misra's language, for example, provides a direct means of expressing an indexed recurrence and its initial conditions [1]. As they readily admit, however, their language does not address the problem of systolic mapping, dealing primarily with formal algorithm development and verification. Combining the language with a mapping methodology [4, 14, 17, 25] could eliminate this problem. Chen and Mead also present a syntactic means of expressing systolic programs as recursive functions [2].

Luk and Jones have developed a functional language for deriving and evaluating systolic mappings [15]. The language can not only define algorithms, but can also define the hardware structure of the circuit implementation. This is particularly helpful for evaluating space-time tradeoffs and connecting subarrays to solve multi-stage problems.

All of these languages can elegantly express a systolic algorithm and its mapping to a processor array. However, they are predominantly systolic design systems — tools for creating, expressing, and evaluating valid systolic designs. As such, they do not deal with the problems of interfacing host and co-processor code, a task which would in most cases require leaving the programming paradigm. Additionally, although these paradigms are excellent for evaluating design tradeoffs, they are not particularly accessible to the general programmer.

The requirements of the New Systolic Language implementation draw from the features of the languages discussed above; it must be emphasized that many of these languages' missions differ from NSL's, and that "faults" can turn into features when viewed from a different perspective. The four guiding principles of NSL are:

1. The language should cleanly separate systolic cell programs and data flow directives. Shared asynchronous variables are to be avoided because multiple references can produce unpredictable results. Instead, cell programs should be specified as pure transfer functions between systolic inputs and systolic outputs.

2. Programs should be able to execute both host and co-processor array functions, preferably in the context of a conventional language.

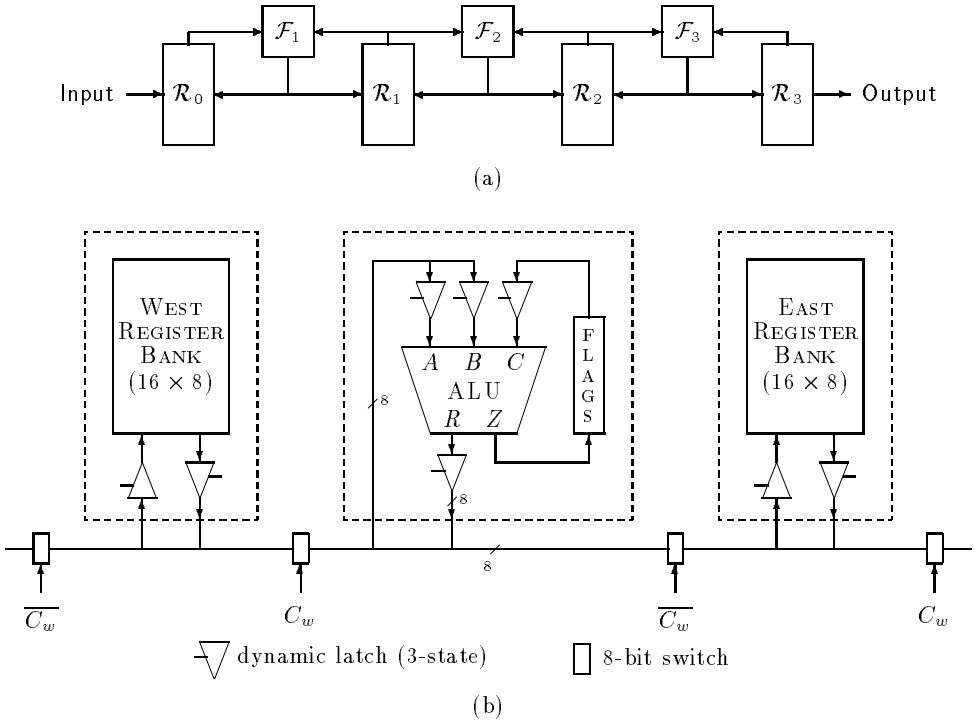

Figure 2: Abstract linear SSRA processor array (a) and the B-SYS architecture (b).

3. The programmer should be able to declare systolic data streams as such using concise flow directives. Stream declaration and initialization should be accomplished apart from the cell program, dissociating cell operation from macroscopic data flow and enabling the reuse of standard data flows and cell operations. Input and output for the systolic streams should be simple and intuitive.

4. The language should be independent of low-level co-processor features and functions, hiding array topology, processing element architecture, array size, and the method of systolic communication from the user. The programmer should not have to specify the physical names of queues, ports, registers, or bits.

3 The Systolic Shared Register Architecture

A previous paper [10] presented the Systolic Shared Register Architecture (SSRA) for programmable systolic computation (Figure 2a). SSRA machines have four distinguishing features: regular topology, SIMD broadcast instructions, shared register banks, and stream oriented communication. The shared registers ($\mathcal{R}$ in Figure 2a) lead to highly efficient systolic communication between functional units ($\mathcal{F}$ in Figure 2a) since computation and communication can be performed simultaneously. For example, the single instruction

$$E_0 \leftarrow \min(W_0, W_1)$$

not only performs a minimization but also moves data eastward. (The registers are addressed relative to the functional units, east or west in the linear case.) The SSRA design readily extends to other topologies, including square and octagonal mesh architectures [9].

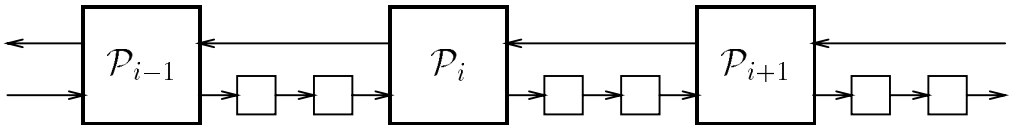

Figure 3: Two systolic streams.

The Brown Systolic Array is an 8-bit linear SSRA machine designed for sequence comparison and other combinatorial applications (Figure 2b). Each 6.9 mm × 6.8 mm 2μ-CMOS chip contains 47 functional units and 48 register banks (85 000 transistors). Each memory bank contains 16 8-bit registers. The B-SYS chips have been fabricated by MOSIS (MOS Implementation Service) and a working prototype with 470 processors and an unsophisticated host to co-processor interface performed 108 million 8-bit operations per second (108 MOPS). A 64-chip version with an on-board instruction sequencer could provide 6 GOPS of processing power, and work is currently underway to design a high speed chip that will provide over 4 GOPS per chip.

4 The NSL Programming Language

The goal of this research has been to design a general-purpose systolic programming environment suitable for a wide variety of machines, from the Connection Machine, which can emulate meshes of arbitrary dimension, to the fixed linear topology of the Brown Systolic Array. The programmer should be able to quickly, concisely, and naturally specify the systolic cell functions and data flows of common systolic algorithms without any knowledge of the target architecture, apart from the fact that it is capable of supporting systolic operations over some broad range of topologies.

Although the Systolic Shared Register Architecture enables very efficient NSL implementation, the architecture is not required by the NSL paradigm. The abstract idea of a systolic stream can be implemented on any parallel processor that either provides or simulates communication between processing elements over a regular network.

As a consequence of being designed for systolic applications, NSL does not support arbitrary programming of individual processing elements. Currently, algorithms that require more than one type of cell program must define systolic streams that distinguish processing elements. It is expected that NSL will be extended to support any constant number of cell programs; for broadcast machines, the several logical instructions streams would be automatically converted to a single SIMD program, while for MIMD machines, all cell programs would be evaluated simultaneously.

This section continues with an overview of the New Systolic Language and a look at NSL cell programs and main programs. Section 5 considers a programming example from computational biology that highlights several advanced NSL features, followed by a summary of the directions in which the language is evolving.

4.1 NSL Overview

The systolic stream is the most important aspect of the New Systolic Language. Conceptually, the systolic stream is viewed as a flow of data buckets passing by the processing elements. During each time step, one bucket in the stream is directly accessible to each processing element. The value contained in the bucket may be used in computation and, if desired, changed. Before the next evaluation of the cell program, this bucket will have

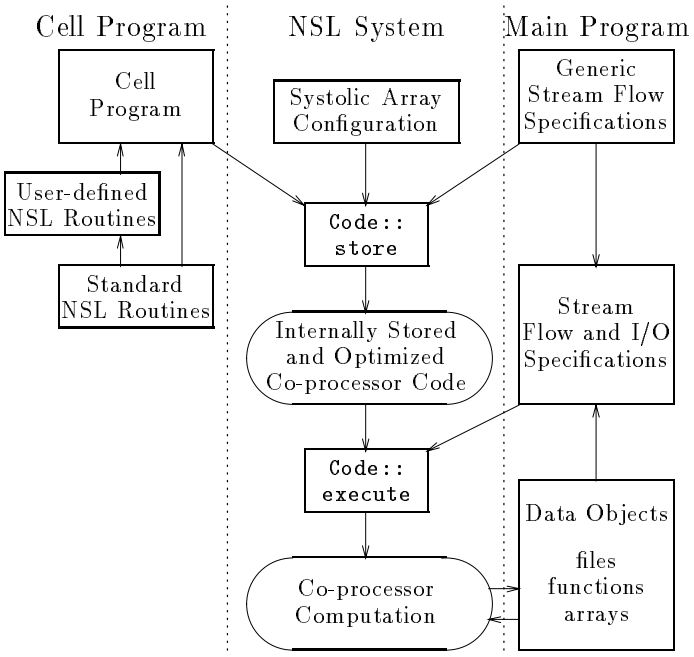

Figure 4: Overview of New Systolic Language programming.

moved downstream to the next processing element or delay register, making room for a new input value.

Systolic speed refers to the number of time steps required for data to travel from one processing element to the next. That is, a stream implements some number of logical delay elements between processing elements which can be represented as shown in Figure 3. The figure diagrams a westward flowing stream of speed 1 and an eastward flowing stream of speed 3. Speed 0 streams are used for immobile and local data which can be automatically preloaded or postcollected when needed. Processing elements can also look small distances upstream and downstream, accessing future inputs and past outputs, as shall be described in Section 5.

Systolic streams are defined in NSL main programs and are linked to the systolic cell program on execution. Systolic cell programs do not concern themselves with stream speed or type: the NSL system assigns registers and performs functions according to the declared type and speed of the systolic stream.

As seen in Figure 4, the NSL system can be divided into three parts: the cell program, the main program, and the NSL system. As mentioned, the NSL cell program specifies computation on abstract systolic data streams, independent of the macroscopic data movement. Cell programs can call user-defined routines which process NSL data types (registers, flags, and data streams) as well as a large number of standard operators and functions. The NSL prototype system was developed in the object-oriented C++ language which provided the ability to overload operators [5]. Thus, common operations have been defined for all basic systolic data types (e.g., when `X` and `Y` are systolic streams, the operation `X+Y` is evaluated by NSL to generate co-processor instructions). C++ has greatly simplified the evolution and evaluation of the language.

NSL routines and cell programs are not called as common C++ routines. References to an NSL cell program are restricted to calls from other NSL routines and cell programs and to use as a parameter to the `Code::store()` procedure (or the `Code::run()` procedure

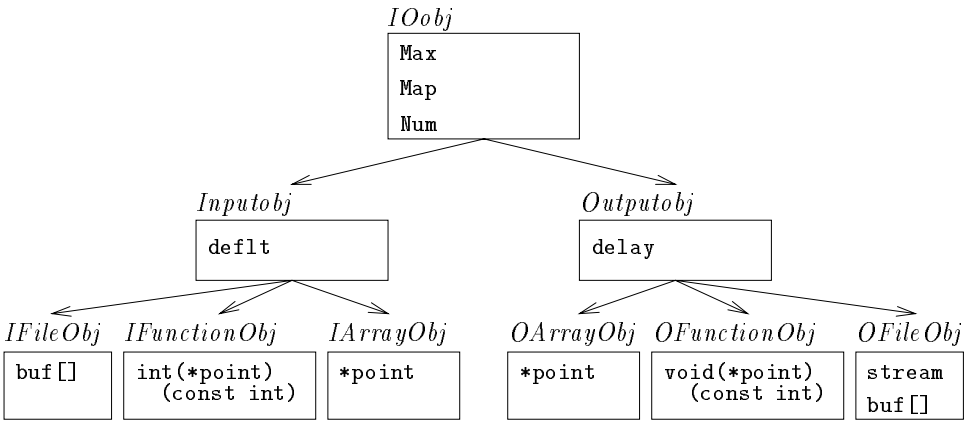

Figure 5: NSL systolic I/O objects.

which combines the functions of storing and executing the code). The NSL system itself will call the cell program and, as a consequence of the overloaded operators, generate code for the systolic co-processor.

Referring to Figure 4, NSL main programs control the flow of information through the systolic co-processor by defining systolic data streams. Generic systolic stream flow specifications include data type (integer or character) and precision information as well as the direction and speed of flow. In the current implementation, flow direction is restricted to eastward and westward, the limits of a linear systolic array; an NSL implementation for mesh systolic arrays would enable several more directions of flow.

In addition to flow information, streams include input and output specifications. Whenever a stream moves through the array, an input is needed and an output produced at the array boundaries. These inputs and outputs can be linked to files, functions, and arrays (Figure 5). By default, the input is zero and the output is ignored.

When the NSL system is initialized, information about the co-processor or simulator configuration (size, topology, and architectural features) is defined. (Much of this information is accessible to the user for the construction of routines that must depend on hardware parameters.) This initialization informs NSL what type of systolic array required by the programmer's application — the NSL system must emulate that architecture if it differs from the actual hardware. A complete NSL system could, for example, automatically execute hexagonal-mesh programs on square- or triangular-mesh co-processors.

After initialization, a call to the `Code::run()` procedure in the user's program will generate (`Code::store()`) and execute (`Code::execute()`) the co-processor code. The `Code::store()` routine detects and resolves potential data hazards, such as overwriting a register needed by a neighboring functional unit, discussed in Section 4.3, and then generates and optimizes code for the co-processor. Using the input and output specifications of the systolic streams, the `Code::execute()` routine sends input to and stores output from the co-processor. As mentioned, the stream I/O specifications can link these inputs and outputs to files, functions, or arrays.

4.2 Cell Programs

NSL cell programs are C++ procedures that make use of NSL's special systolic data objects. No information about data flow, data type, stream inputs and outputs, or array topology

```
#include "nsl.h"
void
horner (SStream& C,
        SStream& X, SStream& Y)
{
  Y = Y + (C * X);
}
```

(a)

```
#include "nsl.h"
void
sort (SStream& Max, SStream& Min)
{
  Flag f = Min > Max;
  Max = select (f, Min, Max);
  Min = select (f, Max, Min);
}
```

(b)

Figure 6: NSL cell programs for Horner's method (a) and sorting (b).

is present in the cell program.

An NSL cell program for Horner's method of polynomial evaluation is shown in Figure 6a.[1] Following the first requirement for systolic programming, this cell program is a pure transfer function that does not involve the macroscopic flow of data; no information about the systolic streams, apart from their formal parameter names, is available to the cell program. A stream name (`Y`) as an rvalue (on the right-hand side of an assignment statement) refers to the input value of that stream, while a stream name as an lvalue (left-hand side) will set the output value of that stream (corresponding to `Y'` in the SDEF systolic design system). Stream names which do not occur as lvalues (`C` and `X`) are given the identity transformation (`C=C` and `X=X`), as with SDEF. If desired, Hearts notation can be used as well, giving direct access to the stream input (`Y.in()`) and output (`Y.out()`) registers. For more complicated access to the streams, several other options will be detailed latter.

Cell programs can use most C++ arithmetic and logical operations and can declare registers or flags for local use, as is done in the NSL cell program for sorting (Figure 6b). The select function is similar to the C programming language's `?:` triadic conditional expression operator, which cannot be overloaded in C++. For more complicated conditionals, several methods of context flag manipulation also exist.

In addition to the predefined NSL operators and routines, users may program new routines involving the NSL data types for use with cell programs, such as minimization routine of Figure 7b. Cell programs and NSL routines can use standard looping constructs to create functions that depend on the target architecture, as illustrated by Figure 7b, an NSL routine for generating a bridge fault test vector. The static *Config* class stores the systolic co-processor's configuration — size, topology, bits per word, and the like. In the future, the configuration will have two parts: the physical configuration and the logical configuration. The NSL system will then be able to simulate a logical architecture on a physical co-processor, though some help from the programmer will generally be required to support this feature.

NSL cell programs only have systolic streams as arguments and never return a value. Data is passed to and from the main program (running on the host) through the systolic data streams according to their input and output specifications. General NSL routines, such as `min()` and `bridge_test()`, can process and return any NSL or C++ data structure.

In summary, NSL cell programs make use of the NSL *Register*, *Flag*, *SStream* and other data types, the normal C++ operators, and a few special functions. The systolic cell programs contain no information about data movement, array configuration, or stream initialization and result extraction; cell programs can be easily reused with different data

[1] In C++, the notation `SStream& C` indicates that the parameter `C` is a reference to an `SStream` object, similar to a Pascal `var` parameter. Constant parameters can be specified with the `const` keyword [5].

```
Register
min (const Register& r1,
     const Register& r2)
{
  return (select ((r1 < r2), r1, r2));
}
```

(a)

```
Register
bridge_test (void)
{
  Register result;
  int i = 0;
  for ( ; i < Config::word; i+= 2) {
    result << 2; result++;
  }
  return (result);
}
```

(b)

Figure 7: NSL routines for minimization (a) and bridge fault testing (b).

flows.

4.3 Main Programs

The NSL calling routine of Figure 8 has full control over the systolic array: it configures the array, maintains the data streams, and calls systolic cell programs as needed. First, the array is initialized to the type (in this case, linear) and size desired. This is followed by several data stream definitions and linkages of stream inputs and outputs. In addition to the basic *SStream* object type, NSL provides several variations. A *FixSStream* is an *SStream* that does not move, and the *SkipSStream*, discussed in the next section, can be used to logically join processing elements.

The NSL system automatically detects data and timing hazards within the systolic streams. When found, an additional register is allocated to the stream and the logical register meanings are *woven* between cell program iterations. For example, the systolic sorting cell program

$$\begin{aligned} E_1 &\leftarrow \min(W_0, W_1) \\ W_0 &\leftarrow \max(W_0, W_1), \end{aligned}$$

which passes the minimum of two values to its eastern neighbor and saves the maximum, has a read-after-write hazard: E_1 is W_1 of a neighboring functional unit, and thus the second instruction will not access the same W_1 value as the first. The solution is to use three registers for the systolic stream, alternating the register which serves for interprocessor communication. Two cycles of this woven cell program would be:

$$\begin{aligned} E_2 &\leftarrow \min(W_0, W_1) \\ W_0 &\leftarrow \max(W_0, W_1) \\ E_1 &\leftarrow \min(W_0, W_2) \\ W_0 &\leftarrow \max(W_0, W_2), \end{aligned}$$

where E_2 and E_1 are alternately used for communication. The use of a local scratch register could solve this problem as well, but would require an additional instruction in each sorting loop. One of the prime functions of the preprocessing `Code::store()` step is to detect these hazards and allocate registers appropriately.

After determining weave conditions, code is regenerated using the revised stream flow specifications and is then compressed to eliminate temporary variables and to combine flag-based and register-based instructions when possible. The resulting cell programs are often as efficient as hand-coded routines and, of course, much easier to understand.

```
#include "nsl.h"
main(void)
{
  int n=5;
    // Array length n, 16 registers, 8 flags, two directions,
    // 1 register bank per PE, linear array, 8-bit word.
  Config::setup (n, 16, 8, 2, 1, LINEAR, 8);

    // Fixed stream of one-byte integer coefficients
  FixSStream Coeff (INT, NSL_BYTE1);

    // Two mobile streams
  SStream X (EAST, NSL_SPEED1, INT, NSL_BYTE1);
  SStream Y (EAST, NSL_SPEED1, INT, NSL_BYTE1);

    // Use a file for input of coefficients,
    // reversing them from the order they appear in the file.
  Coeff.source ("file1", n, REVERSE, 0);
    // Read n X inputs from file2, then default to 0.
  X.source ("file2", n, IDENT, 0);
    // Place n Y outputs in file3 after waiting n
    // steps (Y inputs default to 0)
  Y.sink ("file3", n, IDENT, n);

    // Compile and execute the complete cell program
    // until the Y stream has received n outputs.
  Code::run (&horner, &Coeff, &X, &Y);
}
```

Figure 8: NSL main program for Horner's method.

5 Programming Example

The SSRA and the Brown Systolic Array implementation were greatly influenced (and indeed targeted for) the sequence comparison problems of the Human Genome Project [3]. As the 3-billion-character string of human DNA is transcribed, the availability of fast data analysis tools and co-processors will become critical. The architectural parameters of B-SYS (in particular, bits per word and registers per register bank) were determined in a large part by the requirements of these algorithms. A simple sequence comparison algorithm can illustrate the more refined attributes of a systolic stream.

The edit distance, the number of single-character insertions and deletions required to transform a string a into another string b, both of length n, is the solution $d_{n,n}$ of the recurrence:

$$\begin{aligned} d_{0,0} &= 0 \\ d_{i,0} &= d_{i-1,0} + 1 \\ d_{0,j} &= d_{0,j-1} + 1 \\ d_{i,j} &= \min \begin{cases} d_{i-1,j-1} & \text{if } a_i = b_j, \text{ otherwise} \\ d_{i,j-1} & +1 \\ d_{i-1,j} & +1. \end{cases} \end{aligned}$$

Several more complicated variations of this problem are discussed in the literature [8, 9, 11, 19]. One way of mapping this recurrence to a programmable systolic array is to let each processing element $\mathcal{F}$ correspond to a specific j value, letting $d_{i,j}$ and $d_{i+1,j}$ be computed in $\mathcal{F}_j$ separated by one unit of time. In this mapping, the string b is preloaded into the

```
#include "nsl.h"
#define DEL_COST 1
void
sequence (SStream& Char1, SStream &Char2, SStream &Cost)
{
  // Cost input is d_{i-1,j-1}
  // Cost[-1] is last step's input,  or d_{i,j-1}
  // Cost[+1] is last step's output, or d_{i-1,j}.
  Cost = select (match (Char1, Char2),
                 Cost,
                 select (Cost[-1] < Cost[+1],
                         Cost[-1], Cost[+1]) + DEL_COST);
}
int
iweight (const int n)  // n-th initial weight d_{n,0}
{ return n; }
main(void)
{
  int n = 5;
  int final_cost;
  Config::setup (n, 16, 8, 2, 1, LINEAR, 8);

    // Fixed stream of one-byte characters.
  FixSStream Seq1 (CHAR, NSL_BYTE1);

    // Two mobile streams
  SStream Seq2   (EAST, NSL_SPEED1, CHAR, NSL_BYTE1);
  SStream Weight (EAST, NSL_SPEED2, INT, NSL_BYTE1);

  Seq1.source ("seq1", n);
  Seq2.source ("seq2", n);
    // Use a function for n input values.
  Weight.source (iweight, n);
    // Only save one result (in an array).
  Weight.sink (&final_cost, 1, IDENT, 2*n-1);

  Code::run (&sequence, &Seq1, &Seq2, &Weight);

  cout << "Sequence Distance: " << final_cost;
}
```

Figure 9: NSL cell program for sequence comparison.

array, while the string a systolically flows through the array.

An NSL program for simple sequence comparison is shown in Figure 9. This program illustrates tying functions and arrays to systolic stream inputs and outputs: the function `iweight()` returns the value $d_{t,0} = t$ for each time step t (the index of the time step is passed to the routine when called by NSL). More importantly, the program illustrates advanced manipulation of the systolic stream object type. In the sequence comparison algorithm, three previous results are required: $d_{i-1,j-1}$, $d_{i,j-1}$, and $d_{i-1,j}$, the first two having been computed in $\mathcal{F}_{j-1}$ (at times $t-2$ and $t-1$) and the remaining one in $\mathcal{F}_j$ (at time $t-1$). Since cost data is required from two time steps ago, the cost stream moves at a speed of 2 (thus, three registers in each register bank are automatically allocated to the stream because weaving is required). The output of $\mathcal{F}_j$'s cell program will be $d_{i,j}$, and the input is $d_{i-1,j-1}$. The problem is to access $d_{i,j-1}$, which will be *next* time step's stream input, and $d_{i-1,j}$, which was *last* time step's stream output. In short, $\mathcal{F}_j$ must look upstream and downstream from its current position. In NSL, these values are accessible using array

Function	Meaning
`S`	Stream input (rvalue) or output (lvalue).
`S.in()`	Stream input value.
`S[0]`	Stream input value.
`S[-i]`	Upstream input value.
`S.out()`	Stream output value.
`S[+i]`	Downstream output value.
`S=`	Assignment of stream output value.
`S.source()`	Assignment of file, function, or array as stream's input source.
`S.sink()`	Assignment of file, function, or array as stream's result sink.
`S.force_weave()`	Force S to be woven.

Table 1: *SStream* member functions.

notation, indexing the systolic data stream relative to the current input or output. Thus, the expression `Cost[0]` is equivalent to both `Cost.in()` and `Cost` as an rvalue, all of which refer to the stream input value $d_{i-1,j-1}$. `Cost[1]` retrieves the value one time unit down the systolic stream, or $d_{i-1,j}$ (produced by $\mathcal{F}_j$ last time step). Similarly, negative indices retrieve values upstream, so `Cost[-1]` retrieves the value one time unit up the systolic stream, or $d_{i,j-1}$ (produced by $\mathcal{F}_{j-1}$ last time step). These systolic stream functions are summarized in Table 1.

Note that during each time step, the $d_{i-1,j}$ values are accessed by two adjacent functional units. $\mathcal{F}_j$ accesses $d_{i,j-1}$ as `Cost[-1]`, a value slightly upstream from that functional unit, while $\mathcal{F}_{j-1}$ refers to the same value as `Cost[+1]`, downstream from that functional unit. Because of the shared registers, this value is readily available to both of these neighboring functional units.

Occasionally, processing elements must be logically grouped together to form larger processors with more memory and computational power. The NSL system supports the `SkipSStream` object type to simplify programming such algorithms. This type of systolic stream is initialized according to the desired size of the processor groups. Individual values in the stream are available to all processing elements in a group, and stream output values are taken from the processing element farthest downstream in each group. This data structure has been used with several amino acid sequence comparison algorithms, in which 20-element cost tables corresponding to a specific character b_j are stored across two to four adjacent processing elements (Figure 10), while a_i values use a unit speed `SkipSStream` to travel through the array. During each cycle of the algorithm, each group of processing elements searches the table in *parallel* and the dominant, downstream processing element then computes the next $d_{i,j}$ value. Given a sufficient number of registers in each register bank, a single-processor version could still only process characters at one half the rate of the 4-processor algorithm. Methods and constructs for automatically generating `SkipSStream` objects based on the resource requirements of the user's program are currently under development.

In summary, `SStream` objects are seen as a streams of data flowing past the functional units. Functional units can look short distances upstream and downstream as they compute values to place in in streams. The shared registers of the SSRA design greatly simplify the implementation of this feature.

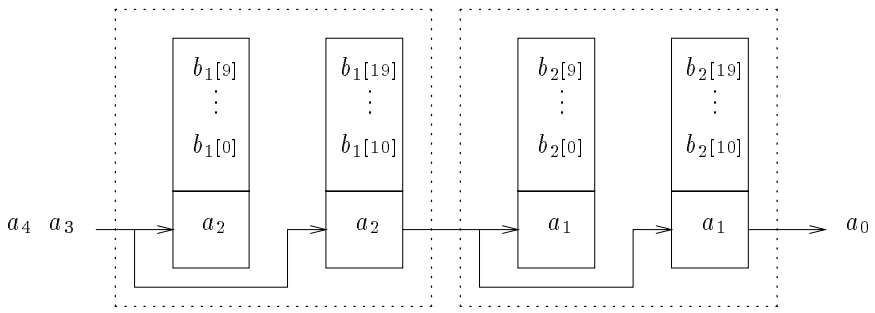

Figure 10: Protein comparison in NSL.

6 Enhancements

The most obvious future development issue is the extension of NSL to topologies of two or more dimensions. In concert with this, stream direction specifications to access arbitrary processing elements within these topologies should be added. A vector notation similar to that of the SDEF system could specify, for example, a stream flowing to the processing element two units north and one unit east. More common flows, such as northwest, would only require textual specification. Depending on actual network topology, some of these streams may require the generation of extra co-processor data movement instructions (i.e., simulating an octagonal mesh on a square mesh). Also, support for two-dimensional input and output specifications, perhaps similar to those of Hearts, should be provided: a systolic stream composed of entire rows from a matrix, distributed along a column of processing elements, should be simple to specify and easy to use.

A possible syntax for a matrix multiplication program is shown in Figure 11. In this mapping of $C = A \times B$, the elements of C are calculated in place, $C_{i,j}$ in $\mathcal{P}_{i,j}$. The A matrix flows eastward in row-major order with $(i-1)$ zeros inserted before $A_{i,1}$. The B matrix flows southward in column-major order with $(j-1)$ zeros inserted before $B_{1,j}$. The entire algorithm requires $n^3 - n^2$ time to complete, at which point the C matrix is read out of the array.

Other future work includes support for wavefront programming. Some algorithms, such as systolic data compression [22], do not have a fixed relationship between input to and output from the array: the delay between input and output depends on how compactly the text can be compressed. Methods for automatically providing sentinels or other mechanisms which implement wavefront programming should be evaluated.

Since NSL is intended for arbitrary systolic co-processors without program modification (apart from NSL initialization via the *Configuration* call), support for oversized and undersized arrays must be developed. In the former case, this could involve controlling a variable-length co-processor, masking certain processing elements, or employing user-guided cell program analysis to find algorithmic methods of coping with large arrays. For example, in the case of sequence comparison, extra processing elements can be loaded with wildcard characters which match all other characters. Thus, by the sequence comparison algorithm, these processing elements will not affect the cost values exiting the array. Similarly, padding strings with null characters will have an entirely predictable effect on the distance computation. Performing functions on undersized arrays is a more complicated, though still achievable, goal [18]. With guidance from the programmer, NSL should be able to partition problems for undersized arrays by storing intermediate results in the host's memory.

```
#include "nsl.h"
void
cell (SStream& A, SStream &B, SStream &C)
{
   C += A * B;
}
main(void)
{
  int n = 8;
  int maxtime = n*n*(n-1);
  int a[n][n], b[n][n], c[n][n];
  Config::setup (n, 16, 8, 2, 1, MESH, 8);

    // Fixed array
    // (default stream type is 2D)
  FixSStream Cstream (INT, NSL_WORD);

    // Two mobile streams
  SStream Astream (EAST,  NSL_SPEED1, INT, NSL_WORD);
  SStream Bstream (SOUTH, NSL_SPEED1, INT, NSL_WORD);

    //  Skew to prepad row i with i-1 zeros.
    //  A source function could perform this as well.
  Astream.source (a, n, n, ROW_MAJOR_SKEW);
  Bstream.source (b, n, n, COL_MAJOR_SKEW);

    // Timed cell program interation needed because the
    // Cstream does not move.
  Code::runtime (&cell, &Astream, &Bstream, &Cstream, maxtime);

  Cstream.sink (c, n, n, IDENT);
}
```

Figure 11: NSL matrix multiplication.

7 Conclusions

The New Systolic Language greatly expedites systolic programming, freeing the programmer from many tedious tasks. As the examples of this paper have illustrated, NSL provides a concise and intuitive interface for systolic co-processor programming. NSL uses the clean separation of cell function and data movement present in the SDEF systolic design system, but additionally provides a common and simple interface to all aspects of systolic programming. The pitfalls of low-level systolic communication directives have been deftly sidestepped by providing automatic hazard detection and avoidance. Additionally, the system is not limited to any particular machine or network topology; future extensions will render the New Systolic Language truly independent of hardware, able to run on a variety of machines and simulators.

8 Acknowledgments

The construction of the Brown Systolic Array was supported in part by NSF grants MIP-8710745 and MIP-9020570, and ONR and DARPA under contract N00014-83-K-0146 and ARPA order no. 6320, Amendment 1. I also thank Daniel Lopresti, Steve Reiss, John Savage, and the anonymous referees for their helpful comments on this research.

References

[1] K. M. Chandy and J. Misra, "Systolic algorithms as programs," *Distributed Computing*, no. 1, pp. 177–183, 1986.

[2] M. C. Chen and C. A. Mead, "Concurrent algorithms as space-time recursion equations," in *VLSI and Modern Signal Processing* (S. Y. Kung, H. J. Whitehouse, and T. Kailath, eds.), ch. 13, pp. 224–240, Englewood Cliffs, NJ: Prentice-Hall, 1985.

[3] C. DeLisi, "Computers in molecular biology: Current applications and emerging trends," *Science*, vol. 246, pp. 47–51, 6 Apr. 1988.

[4] V. V. Dongen, "Quasi-regular arrays: definition and design methodology," in *Systolic Array Processors* (J. McCanny, J. McWhirter, and J. Earl Swartzlander, eds.), pp. 126–135, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[5] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.

[6] B. R. Engstrom and P. R. Capello, "The SDEF systolic programming system," in *Concurrent Computations* (S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds.), ch. 15, pp. 263–301, New York: Plenum Press, 1988.

[7] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer*, pp. 26–40, Jan. 1980.

[8] M. Gokhale *et al.*, "Building and using a highly parallel programmable logic array," *Computer*, vol. 24, pp. 81–89, Jan. 1991.

[9] R. Hughey, *Programmable Systolic Arrays*. PhD thesis, Dept. Computer Science, Brown University, Providence, RI, 1991. Tech. Rep. CS-91-34.

[10] R. Hughey and D. P. Lopresti, "B-SYS: A 470-processor programmable systolic array," in *Proc. Int. Conf. Parallel Processing* (C. lin Wu, ed.), vol. 1, pp. 580–583, CRC Press, Aug. 1991.

[11] S. Karlin and S. F. Altschul, "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes," *Proc. Natl. Acad. Sci. USA*, vol. 87, pp. 2264–2268, 1990.

[12] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[13] M. S. Lam, *A Systolic Array Optimizing Compiler*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1989.

[14] P. Lee and Z. Kedem, "Synthesizing linear array algorithms from nested FOR loop algorithms," *IEEE Trans. Computers*, vol. 37, pp. 1578–1598, Dec. 1988.

[15] W. Luk and G. Jones, "The derivation of regular synchronous circuits," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 305–314, IEEE Computer Society, May 1988.

[16] D. May and R. Taylor, "OCCAM—an overview," *Microprocessors and Microsystems*, pp. 73–79, Mar. 1984.

[17] D. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Trans. Computers*, vol. 31, pp. 1121–1126, Nov. 1982.

[18] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Computers*, vol. 35, pp. 1–12, Jan. 1986.

[19] D. Sankoff and J. B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Reading, MA: Addison-Wesley, 1983.

[20] L. Snyder, "Hearts: A dialect of the Poker programming environment specialised to systolic computation," in *Systolic Arrays* (W. Moore *et. al.*, eds.), pp. 71–80, Boston, MA: Adam Hilger, 1987.

[21] J. T. Stasko, "Tango: A framework and system for algorithm animation," *Computer*, vol. 23, pp. 27–39, Sept. 1990.

[22] J. A. Storer, *Data Compression*. Rockville, MD: Computer Science Press, 1988.

[23] Thinking Machines Corporation, Cambridge, MA, *Paris Release Notes (version 5.1)*, June 1989.

[24] J. A. Webb, "Steps toward architecture-independent image processing," *Computer*, vol. 25, pp. 21–31, Feb. 1992.

[25] Y. Yaacoby and P. R. Capello, "Scheduling a system of affine recurrence equations onto a systolic array," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 373–382, IEEE Computer Society, May 1988.