NuFloW: A Programming Environment for the NuMesh Computer

by
Philippe P. Laffont

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment for the degrees of

Bachelor of Science
and
Master of Science

at the
Massachusetts Institute of Technology
September 1, 1991

Signature of Author: Signature redacted
Department of Electrical Engineering and Computer Science
September 1, 1991

Certified by: Signature redacted
Stephen A. Ward
Professor, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Signature redacted
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

NuFloW: A Programming Environment for the NuMesh Computer

by
Philippe P. Laffont

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment for the degrees of
Bachelor of Science
and
Master of Science

Abstract

The NuMesh project is an attempt to develop packaging, interconnect, and communication technologies that will enable to connect multiple processing elements together. Performance will be achieved through very high communication bandwidth. The motivating philosophy behind NuMesh is to provide very simple but extremely fast hardware, and manage the complexity of the system through a set of sophisticated software protocols and tools. Currently, the NuMesh uses a static routing model.

Writing programs on the NuMesh is a complicated process. The goal of this thesis is to present NuFloW, a graphical programming language/user interface, which lets users build programs by assembling graphical blocks together and connecting them with streams.

Acknowledgments

First and foremost, I would like to thank my advisor, Steve Ward, for numerous helpful conversations, both with respect to my thesis, and a multitude of other subjects. I am not sure I would have found the inspiration and energy to finish without his help.

I would also like to acknowledge Olivier Blanchard, who made me realize there was at least another field as interesting as Computer Science. In addition, without him, I would not have found another person willing to sign all my petitions.

Finally, I would like to thank a few good friends, Krin, José, Edan, Leo, and Manuel, and of course my family, who must still wonder what is so fascinating about computers.

The Scholars

Bald heads forgetful of their sins,
Old, learned, respectable bald heads
Edit and annonatate the lines
That young men, tossing on their beds,
Rhymed out of love's despair
To flatter beauty's ignorant ear.

All shuffle there; all cough in ink;
All wear the carpet with their shoes;
All think what other people think;
All know the man their neighbour knows.
Lord, what would they say
Did they Catullus walk that way?

– William Butler Yeats, 1915

Contents

Chapter 1

NuMesh

1.1 Introduction

Backplane buses [10], such as NuBus, have enjoyed much popularity as an engineering discipline to build modular systems: bus interconnection techniques allow system components to be designed independently, without affecting the rest of the modules. However, the limitations of the backplane bus are well-known: Multiple transactions cannot happen concurrently, the size of the bus affects the maximum clock rate at which it can be timed, and buses offer finite expansion since they are not scalable. As a result, large-scale multi-processor machines (BBN Butterfly [2], Cosmic Cube [15], Connection Machine [4], iWarp [13]) have developed their own multi-dimensional buses to support high-bandwidth communications between processors. However, these buses are specific to the hardware they run on, make a number of routing policy decisions at the hardware level, and thus are not of much use to the digital designer.

The NuMesh project [18, 19] is an attempt to provide a standardized communication substrate that permits connecting an arbitrary amount of processing elements together. NuMesh proposes to decouple the communication interface from the processing elements, allowing the design of a highly optimized, modular and cost-effective system. In this way, the digital designer can interface a module of his choice to NuMesh, without needing any prior knowledge of the gestalt of other modules which may populate the mesh.

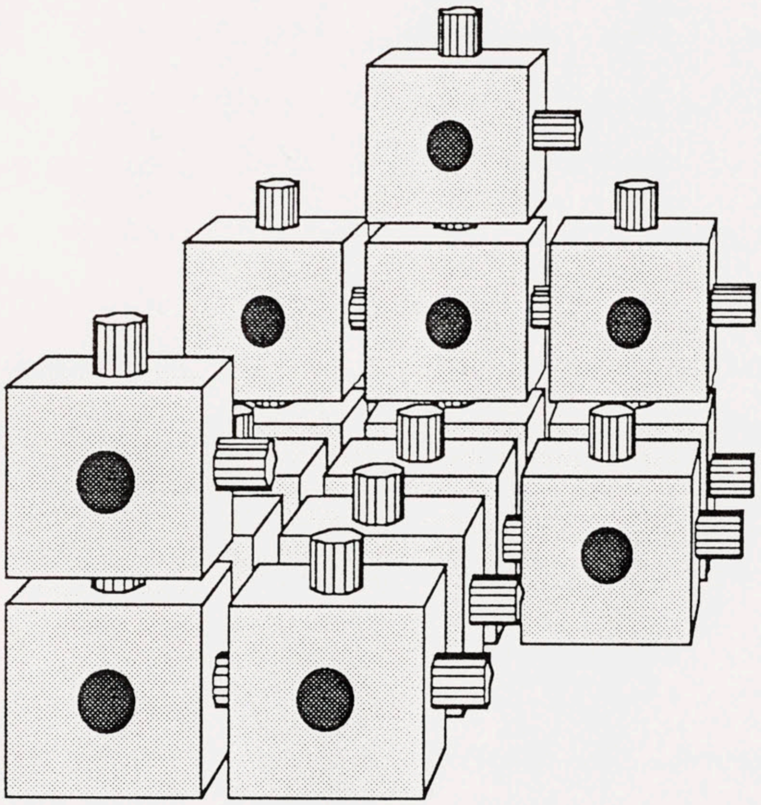


Figure 1.1: Artist's Conception of a NuMesh Computer.

The motivating philosophy behind NuMesh, much like that of RISC computers, is to provide very simple but extremely fast hardware, and manage the complexity of the system through a set of sophisticated software protocols and tools. In a sense, NuMesh is just a network of wires and registers, a very large data-path which connects multiple elements. Absolutely no routing policies are embedded in the hardware. The software is free to choose one of numerous routing models.

NuMesh uses a static routing model. Static routing offers an enormous speed-up potential: (1) Processors do not idle for data because they know the exact time of arrival of each message. (2) No time is spent decoding/encoding messages; the messages are smaller (no headers) and all of the network's bandwidth is used to carry useful information. (3) Without dynamic routing, the messages can potentially travel at the speed it takes an electrical signal to travel from the source to the destination node. (4) Static routing avoids dealing with deadlock. Because all the data transfers are predicted at compile time, the scheduler can reorder messages to avoid deadlock. NuMesh makes provisions to allow for some dynamic routing, but emphasizes predicting most of the communication patterns at compile-time and scheduling them statically.

The multi-grid algorithm was tested on NuMesh and Alewife [1, 12] (a scalable, shared memory multi-processor machine being built at MIT). Preliminary results show that the static routing of the NuMesh out-performed the dynamic routing of the Alewife machine by a factor of up to 40.

The comparaison was biased in many ways, and NuMesh does not expect to outperform other machines by such a factor.

1.2 NuMesh Hardware Overview

Conceptually, the NuMesh is a 3-dimensional lattice which connects nearest neighbors. Each module, of approximately two cubic inches, communicates directly with all six of its neighbors (figure 1.1). Data paths are unidirectional; two neighbors can exchange information in both ways during a clock cycle. Power and cooling are also provided through each of the cube's sides.

The current prototype is built from off-the-shelf TTL parts and is two-dimensional (to sidestep cooling and layout issues). Each node is made up of two separate components: a Communication Finite State Machine (CFSM) and a Processing Element (PE). CFSMs run at 25 Mhz, and route the data around the network. Routing is pre-programmed into each CFSM. The processing element uses a Texas Instrument TMS320C30 digital signal processing chip running at 27 or 33 Mhz and contains 8 Kwords of SRAM. The PE does all the computing and receives/sends data to the CFSM through two FIFOs which reside on the CFSM. The FIFOs enable the PE to run asynchronously from the mesh. Each processing element generates its own clock. The mesh (i.e. CFSMs) uses a globally synchronous clock. The method to deal with clock skew and fan-out is primitive: the clock is distributed to all CFSMs by matching wires of the same length and impedance. A Macintosh II workstation drives the NuMesh and service I/O requests.

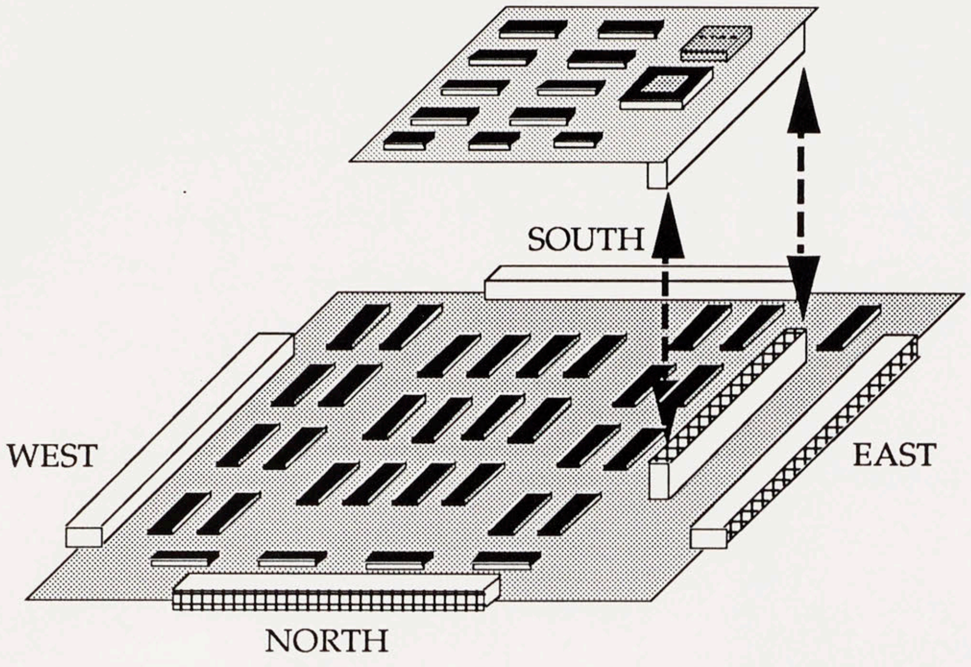


Figure 1.2: CFSM and PE of a current NuMesh module.

Clock skew and fan out should not be an issue in the next NuMesh prototype. NuMesh will use an adaptive clock synchronization scheme, where each CFSM generates its own clock, and synchronizes it by comparing its skew with its immediate neighbors [9]. Redesign of the CFSM [5, 14] and processing element [16] should provide a NuMesh computer running close to 150 Mhz, and processing elements using a 40 Mhz SPARC chip, with 16 Megs of DRAM.

1.3 NuMesh Software Overview

The current programming environment lets users write code for the NuMesh in a high-level language, C. But producing executable binaries is a complicated process involving several host computers [8]. A compiler must generate code for each of the processing elements. A scheduler then maps and pre-compiles the communication between all the nodes.

The C programs must adhere to a specific format composed of an initialization loop and a main loop which is repeated forever. The code is compiled down using Texas Instrument's C30 compiler. The C30 compiler was chosen because it is highly optimized to take advantage of the C30 capability to execute instructions in parallel. The compiled result is then fed through a linker, which maps the various segments to the appropriate memory locations of the C30.

The CFSM code is automatically generated by a scheduler. Internode communications are word based. Nodes refer to each other by names. All communication must be predictable: Flow Control statements must contain the same communication patterns for all possible outcomes, and cannot depend on any variable. In the first step of the scheduler, the C modules are simulated and all of the communications paths are recorded. In the second step, the scheduler maps all of the communications in a way that prevents resource contention.

A Macintosh computer, serving as a front-end to the NuMesh, is able to sense the topology of the mesh. During the loading process, a tree of all nodes is formed, and appropriate CFSM code is generated to load a CFSM and a DSP file for each node involved in the process. The nodes are loaded from the back to the front. When all nodes are finished loading, execution starts.

1.4 Limitations of the Current Software

The previous discussion highlights several problems with the current NuMesh prototype, both at the hardware and software level. These limitations are important to understand because they affect the programming model in several ways. The next two sub-sections examine these problems, and section 1.4 introduces NuFloW, a new language/user interface, specifically tailored to the NuMesh environment, to alleviate the problems discussed below.

1.4.1 Hardware Limitations (from a software designer's view)

Currently, a CFSM cannot wait for a number of cycles before resuming execution. CFSMs frequently need to wait for a datum from one of their neighbors, possibly for a few cycles. Indeed, data communications cannot be perfectly pipelined. As a result, a large number of CFSM states are consumed by `NOOP` instructions. In fact, simulating the multi-grid algorithm on large NuMesh configurations, over 90% of the CFSM states where `NOOPS` [12]. Adding a counter to the CFSM would enable it to wait for a fixed number of cycles. The counter should be addressable from the CFSM and hopefully from the PE.

The CFSM should also be able to make some basic flow control decisions, such as `if` and `goto`. This would further contribute to diminishing the number of states required by each CFSM program. Again, a counter would probably be sufficient, which would also offer the PE the possibility to control CFSM state of execution. These problems will be solved in the next CFSM revision.

1.4.2 Software Limitations

Writing code for the NuMesh is a difficult process. The hardware limitations are reflected in the software:

• The mechanism to send and receive data from one element to another reflects the word-based data transfers between all CFSMs. High level procedures should let the user send messages of arbitrary length. The message could be sent in consecutive cycles or could be typed with some ordering information and sent in chunks, a decision left to the scheduler. It is even possible to envision providing the user with a wide class of messages: compressed, encrypted or acknowledged. In addition, messages could be sent point-to-point, broadcasted, or diffused.

• When the user writes code which involves a control flow decision, he must insure that all branch outcomes generate the same pattern of communication. Although padding the various branches may be inevitable (until dynamic routing at least), the user should not have to enforce these conditions. The compiler could easily generate this information.

• A traditional language like C does not reflect Numesh's static routing limitations. Flow [17] was an initial attempt to embed in the language certain constructs that would underline Numesh's strengths and weaknesses. Unfortunately, Flow was difficult to use; it tried to replicate much of the functionality of traditional textual languages, and few programs were written using it.

• A large class of applications well suited to multiprocessor machines like NuMesh, replicate the same code among many nodes. It is thus of vital importance to be able to run the same code on every node. With the current set-up, identical tasks running on different processors cannot share the same source code. At best, through the repeated use of clever pre-processor macros, the user can avoid writing code for each module. This method is self-defeating because it requires in-depth knowledge of the CFSMs.

• There is no easy way to specify compile-time and run-time parameters. In many real-time applications, the user needs to tweak some parameters on the fly, without having to recompile.

• Current NuMesh C code is written as a whole, on a per-processor basis. It is thus very difficult to split this code into modules which can be run independently on separate processors.

1.4 The Role of NuFloW

The above discussion presents compelling evidence for designing and implementing a new language/user interface tailored to the NuMesh. The rest of this thesis introduces NuFlow, a graphical language and user interface, which lets users build programs by assembling graphical blocks together and connecting them with streams. The role of NuFloW is to abstract out the communication details from the PE code. A user will write blocks of code in the same manner, whether they run on same or separate processors. NuFloW will restrict the user from writing code which is difficult for the NuMesh to run, and will also provide libraries of pre-compiled optimized code, which the user can easily incorporate into his program. Finally, NuFloW will let the user rapidly prototype code in a unified environment.

NuFlow does not try to supercede a language such as C. Rather, it complements C by offering various mechanisms to solve some of the problems previously identified. NuFloW is not a universal language, and it is as much a user interface as a language. NuFlow is a front-end, and as such, it will rely on compilers and schedulers to produce executable code. NuFloW will not provide timing information to the various compilers. As a result, CFSM code will still need too be hand-tuned for real-time applications which require enormous throughput. Indeed, the scheduler needs to make very conservative decisions because it is not provided any timing information during the I/O traces and the CFSM code it generates is not very efficient. Eventually, the NuMesh compiler technology will be sophisticated enough to provide traces which include timing estimates [3].

Chapter Two provides a detailed description of NuFloW. Chapter Three shows how to use NuFloW to build a complex program. Chapter Four discusses some of the important issues concerning the NuMesh programming environment, and contains recommendations for future work.

Chapter 2

NuFloW: Language specification

2.1 Introduction

NuFloW is a pictorial language where tasks are depicted by graphical symbols, called *blocks*. Blocks are hierarchical; complex blocks can be assembled from simpler ones. Blocks exchange information through *ports* and *streams*. A stream connects two blocks through ports, and carries information between them. A block can have any number of ports. It is important not to think of streams as physical or even virtual (in the sense of being a multiplexed physical connection) channels between blocks. A stream displays internal as well as external communication among processing elements.

NuFlow was designed as a pictorial language for several reasons. First, it is intuitive to manipulate and assemble blocks together, much like playing with Legos or TinkerToys. Second, moving from a one-dimensional space with textual languages to a two-dimensional space with a graphical language tends to make the parallelism more explicit. Third, the NuMesh is well modeled by a block + stream based paradigm. Indeed, streams create a fairly controlled flow of communications, well-suited to the static routing of the NuMesh. Fourth, and most important, at the user interface, streams abstract away communication details from the contents of a block. Consider the following trivial computation, `y = f(g(x1), g(x2))`, represented alternatively in C and NuFloW code.

```
int main()
{
   int x1, x2, y;
   int temp1, temp2;

   x1 = read_NuMesh();  /* read values from the mesh */
   x2 = read_NuMesh();
   temp1 = g(x1);
   temp2 = g(x2);
   y = f(x1, x2);
   return y;
}
```

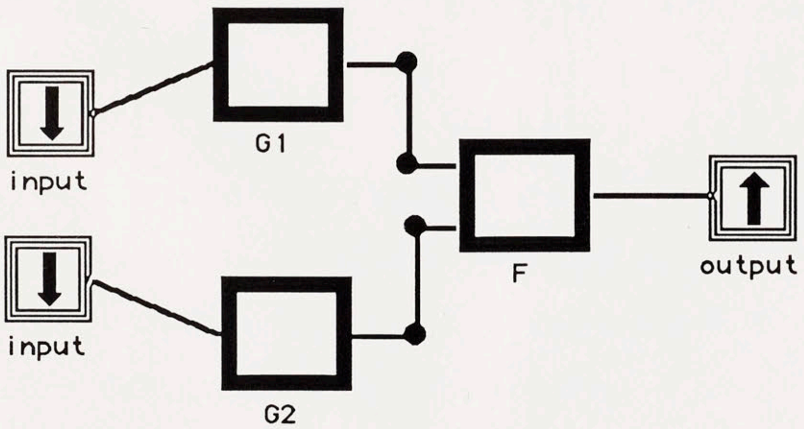


Figure 2.1: NuFloW sample code

The C code is perfectly functional, but assumes it runs on a single processing element. Thus, it needs a re-write to run on two or more processors. The NuFloW code uses C to provide primitive operations, but does not make any assumption on the number of processors which might be used in the computation of `f`. As a result, `f`, `g`, and `h` could be mapped onto one, two or three different processors, without having to re-write a single line of code.

NuFloW relies on external textual languages to perform all of the computation. Many graphical languages/user interfaces supply all the functionality of the processor at the graphics level, and alienate the programmer by cluttering his worksheet with icons, and making him learn all the intricacies of yet another language. Text is a more efficient medium to manipulate constants and primitive operations. NuFloW, however, does not make any assumption on the language used. The above example uses C for illustrative purposes. But in fact, every block defined in NuFlow could contain a module written in a different language. In addition, external languages let the user benifit from the installed base of tools written for that language.

Section 2.2 through 2.5 give precise definitions of blocks, ports, and streams respectively. Section 2.6 introduces NIFT (NuMesh Interchangeable Format for Text), which provides a standardized representation of text between NuFloW and the NuMesh compilers and schedulers.

2.2 Blocks

A block corresponds to a sequence of tasks to be performed on the NuMesh and does not specify the hardware it runs on. For instance, CFSM and PE tasks are just blocks expressed in different source languages. A block can even represent a specialized module (I/O board, microphone, speaker etc..) which performs a task at the hardware level.

2.2.1 Block characteristics

• A block has a name. Each block must have adifferent name.

• A block has a code definition, which is a file that contains the definition of the task represented by the block. The code definition could be a NuFloW file, in which case the block is hierarchical, or a source language file. The code definition implements the instance/class distinction. All blocks which share the same code definition are instances of a particular class.

• Finally, a block contains some textual information, describing the use of each port within the block (the NuFloW user is encouraged to use this field profusely, especially if he plans to incorporate blocks into a library).

2.2.2 Port characteristics

A block imports and exports data through ports. A block contains as many ports as necessary.

• Ports can be read-only, write-only, or read-write. At the user interface, all ports appear similar. The read-write privilege information is meant to be used by a type-checker (see section 2.3.2). In C for example, a user could send data to another block with the construct:

```
write_port(port_out, &foo, sizeof(struct foo));
```

and receive data with:

```
read_port(port_in, &foo, sizeof(struct foo));
```

• Each port is identified by a *port number,* which is unique to the block but not the whole set of blocks. Not only is the space for port number local, but it is fixed. This enables instances of a same block to share the same code without the intervention of a post-processor to bind ports to the appropriate names. To fix the name space, NuFloW enforces that blocks be numbered in a sequence from 1 to n, where n = total number of ports. The user can add, delete or renumber ports, but blocks must always be numbered consecutively.
• A port has a name, but it is optional and only used at the user interface level.

2.3 Streams

Two blocks exchange data through streams. Streams are mostly a user-interface artifact. They remind the user how blocks interact with each other. Two streams cannot connect to the same port. If streams need to be merged, split or interleaved, the programmer should use a block (CFSM code) to perform the required function, and connect the resulting streams appropriately. However, one stream could connect to two ports of a same block, in effect forming a recursive block. Recursion is difficult to implement in a static routing model. Some recursions, such as an iterative process which converges to an answer and then halts, can be computed by the NuMesh, and therefore NuFloW does not forbid recursive blocks.

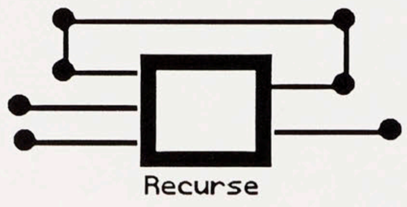


Figure 2.2: Recursive block with 3 inputs and 2 outputs.

2.3.1 Stream Characteristics

• A stream has a name, but it is optional, and only used at the user interface level.

• A stream is composed of 2 endpoints and a number of articulations in between. The purpose of articulations is visual. Endpoints connect to ports.

• One endpoint is labeled source, the other destination. Yet, the labeling does not imply a direction of flow of data. The port's read-write information specifies the data flow.

2.3.2 Stream type-checking

In theory, stream could carry different data types through the course of a computation. This flexibility, however, would lead to serious type-checking problems. Consider the following trivial pseudo-C scenario:

```
/* block 1 */
write_port(port_out, &foo, sizeof(foo));
write_port(port_out, &bar, sizeof(bar));

/* block 2 */
read_port(port_in, &bar, sizeof(bar));
write_port(port_in, &foo, sizeof(foo));
```

Block 2 reads information sent by block 1, but in the wrong order. Such code would undoubtedly cause errors, and methods must be designed to prevent such problems. NuFloW's philosophy is to embed as few constraints as possible at the user-interface level, and to let a variety of software tools do the type checking. This solution does have drawbacks. If block 1 and block 2 are written in the same source language, writing a type-checker is easy. But if block 1 and block 2 are written in different languages, it would be very difficult to deduce any type-checking information. Furthermore, catching these errors is vital, because a type-checking error in one block may propagate through a whole series of blocks, and cause bugs far from the original location of the error. Another reason to add some type-checking information to NuFloW stems from the fact that different NuMesh modules may have separate internal representations for primitive types (little vs. big endian), and the compiler need not derive this information from the streams. This problem could actually be solved by stipulating that all messages on the NuMesh follow a little or big-endian convention, but it would be inefficient. The above argues for NuFloW to act as a common denominator type-checker, raising the question of writing a universal type-checker, without hindering the programmer.

NuFloW's solution is a compromise. A user can choose to add type information to a port, but it is not mandatory. Each type is represented by a string. A default type library is loaded when NuFloW is executed. For instance, the C type library would contain: "int", "long", "void", "double".. To declare a type for struct foo, the user would just add the string "struct foo" to the port information. A user could add more types to a port if he needed the corresponding stream to carry multiple types of data.

2.4 NIFT

NIFT [11] (NuMesh Interchangeable Text Format), provides a standardized representation of text between NuFloW and the NuMesh compilers and schedulers. NIFT is designed to keep files as compact as possible. A NIFT file is composed of two parts: a header, which provides information to the compilers and scheduler, and a second part, which describes of all the NuFloW blocks. There are three types of NuFloW blocks: node blocks, leaf blocks, and a root block. The next sections contain a detailed description of each component of a NIFT file.

2.4.1 NIFT syntax

- Strings are enclosed by double-quotes.
- The @ character is a special character and cannot be used in any source file, or as part of a block, stream or port name. To use the @ character, use \@ instead.
- A sequence of ;;; is used to add comments to a NIFT file.

• Information is indented in a LISP-like (parentheses) format.

2.4.2 NIFT Header

The header provides programming paradigm information, NIFT version number and other useful information. All text files used in the NuMesh world should adopt the UNIX convention to use the first line to describe the file.

```
;;; - this is a NIFT file -
(header
   (paradigm "stream")
   (NIFT-version 0.1)
   (date <date>)
   (blocks <total block number>)
   <optional information>
   ..)
```

2.4.3 Node Block

A node block contains sub-blocks and is defined as follows:

```
(block
   (name <block name>)
   (ports
      (<port1 rw-priv> <child-block name> <child-block port>)
       ..)
   (sub-blocks
      (<block-name> <code-definition>)
       .. )
   (streams
      ((<block-source> <port-source>)
       (<block-dest> <port-dest>))
      ...))
```

• The name field contains the block name. Each block must have a different name.

• The `ports` field contains a list of all of the node block's ports and a description of how they "hook-up" to the streams in the code definition file. Because ports are automatically numbered by NuFloW, it is unecessary to include their name and number.

• The `sub-blocks` field contains a list of all sub-blocks' name and code definition. The code definition field insures that only one copy of the code definition file will ever be referenced. Indeed, if a block contains multiple instances of a particular function, then although each instance may have a different name, it has the same code definition. All blocks use a single global name space, which easily leads to block name aliasing. This problem, however, can be solved by using blocks' *pathanmes* instead of names (a block pathname is defined as the name of a blocks preceded by the name of all of its ancestors back to the root block). In fact, the name space of blocks should mirror that of the file system which NuFloW runs on top. For technical reasons (the Macintosh's unconventional and opaque file system), the current implementation of NIFT uses a single, flat, global name space. Also note that a block could have no code-definition, in which case it would be assumed that the processing element/CFSM the block eventually gets mapped to NuMesh nodes that will automatically produce/consume data.

• The `stream` field is a list of all streams connecting sub-blocks. The source and destination are specified by a block and port number. Streams which connect to argument blocks are not included in this section, since that information is conveyed in the `ports` field.

2.4.4 Leaf block

A source block contains source code instead of NuFloW code.

```
(block
   (name <block name>)
   (ports (<port1 rw-priv> <port2 rw-priv> ..))
   (source <code-definition>))
```

The first two fields of a leaf block are identical to a node block. The third field contains the code-definition filename.

2.4.5 Root Block

The root block is a syntactic construct that groups information about the NufloW project to let the compiler know where to start. The root block is build from the top-level NuFloW file supplied by the user.

```
(block
   (name @root-block@)
   (ports)
   (sub-blocks <name1> .. <name n>)
   (streams ((<block source> <port source>)
             (<block dest > <port dest>)
            ...))
```

The root block has no ports, and the name is fixed. At the user interface level, the root block is never displayed.

Finally, NIFT files could be made significantly smaller by replacing all names with integers. The strings are used for debugging purposes, and the NIFT format will change when the compilers are running.

Chapter 3

NuFloW Programming Example

3.1 Introduction

This chapter serves as a case study of the NuFloW language. It shows the feasibility of NuFloW as a programming environment for the NuMesh, and will be referenced by other chapters to illustrate both important design decisions and problems/improvements with the current NuFloW implementation. The next section gives a brief overview of the NuFloW editor. Section 3.3 contains NuFloW code for a spectrogram demo running on the NuMesh. Section 3.4 describes the NIFT representation of the spectrogram code.

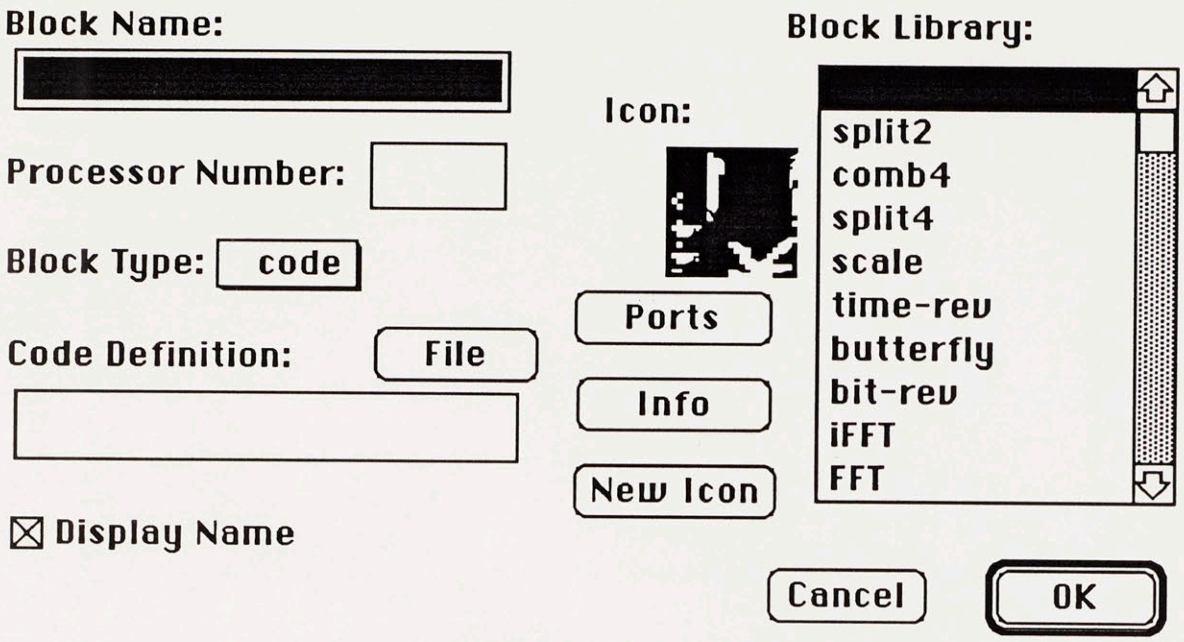
3.2 NuFloW editor

The current NuFloW editor runs on the Macintosh computer. The Macintosh platform is used because of its user interface, and because it serves as the front-end to the current NuMesh prototype. Undoubtedly, a UNIX workstation will be used to drive NuMesh computers in the future. Therefore, the editor was designed to be as modular as possible, and should be readily portable to another graphical interface such as X Windows.

NuFloW, much like a conventional text editor, does not work on a particular project. All files are independently edited. Files can be compiled at any time. All work is done on worksheets. An unlimited number of worksheets can be opened (RAM is the only limiting factor).

3.2.1 Block editor

Blocks are created using a block editor. The block editor contains a list of blocks loaded from a default library. Each newly created block is an instance of the corresponding block in the library. Of course, it must be possible to create new blocks in addition to instances of pre-defined blocks, and the user has two choices: 1) the user can edit an instance, and change its attributes. 2) the user can load a new icon (from a resource file), and create a new block depicted by that icon.



Block Name:

Processor Number:

Block Type: code

Code Definition:

☒ Display Name

Block Library:

split2
comb4
split4
scale
time-rev
butterfly
bit-rev
iFFT
FFT

Figure 3.1: Block Editor

• Each block in a worksheet must have a different name. NuFloW will not permit two blocks with identical names to be displayed in the same worksheet.

• The code definition field specifies the file to open when the block is "double-clicked". The file could be a text or graphics file.

• The processor number field specifies which processor the block should run on. The number is made of four digits, the first two specify the horizontal coordinate of the processor in the mesh, and the next two the vertical coordinate. Eventually, the NuFloW and NuMesh loader will be merged, and the user will drag blocks onto NuMesh nodes to specify processor numbers. Only top-level blocks contain processor information. All other blocks run on the processor assigned to their top-level ancestor. In the long-run, the user will not need to specify this information, as the NuMesh compiler will automatically figure out the best configuration to run on.

• The new icon and load button are used respectively to load new icons from a resource file, and blocks from a NuFloW graphics file.

3.2.2 Ports editor

Streams connect blocks through ports. A block contains as many ports as desired. All streams must connect to different ports though. The port editor lets the user:

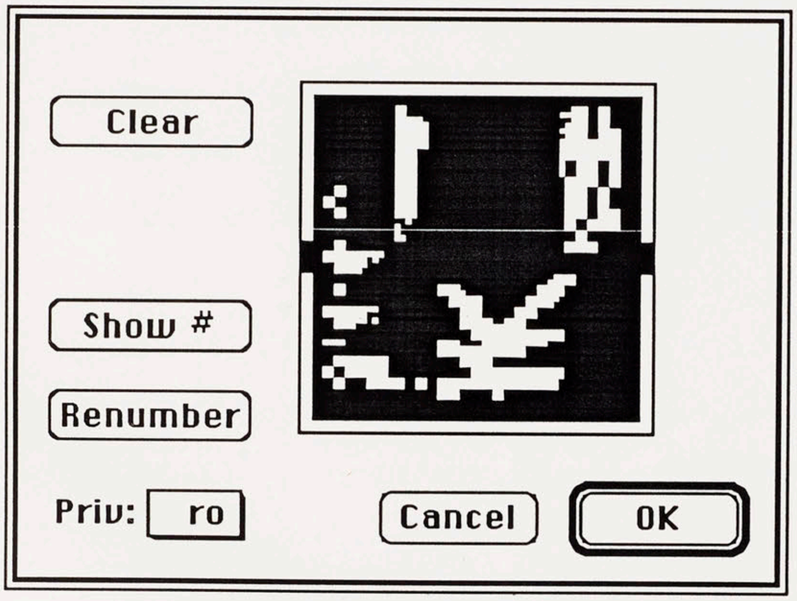


Figure 3.2: Port Editor

• add ports by clicking on the desired location of the port.

• delete ports by clicking on the location of the port to remove.

• renumber ports: All ports are consecutively numbered. Instead of moving streams around to change argument bindings, the user can simply renumber ports to achieve the same result.

3.3.3 Streams

Streams remind the user how blocks communicate to each other. Streams are only present at the user-interface, and are not included in the NIFT representation of a project. Streams can be routed around blocks by adding articulation points.

3.3.4 Text Editor

The current NuFloW text editor is primitive and should only be used to verify the contents of text files references by leaf blocks. As the new Macintosh system 7.0 gains popularity, it will be possible to interface a better text editor through the Apple-event mechanism. For the present time, users are encouraged to use a word processor suited to editing programs.

3.3.5 Matching Inputs and Outputs to Sub-blocks

Streams join blocks within the same worksheet and cannot reference external blocks. At the user interface though, a mechanism must be provided to match inputs and outputs to the appropriate ports of the parent block. A special class of blocks, called argument blocks, is used for this purpose. An argument block has no code definition, and exactly one port (the user can change its location). In addition, the argument block has an extra field, which indicates the parent's port number it is supposed to represent.

3.3.6 Libraries

When the NuFloW application is launched, an default library of blocks is loaded in memory. Once a block is loaded, it cannot be modified. To add a block to a library , simply open the library file, add the block, and save the file. Changes in a library are not reflected until the next NuFloW session. In fact, library files are no different from regular graphic files, and any file can be used as a library. Finally, the user can load multiple libraries in memory, and change the default library.

3.3 Spectrogram

The purpose of this section is to show how to use NuFloW to build a spectrogram that could run on the NuMesh. Spectrograms are an important component of voice recognition systems. The spectrogram demo converts a sound stream from time to frequency domain, using a iterative FFT algorithm. Spectrograms are well-suited to the NuMesh because they repeatedly perform the same computation, which accelerates linearly with the number of processors employed.

3.3.1 Spectrogram Pseudo-Code

The spectrogram code is based on an iterative FFT:[1]

```
ITERATIVE-FFT(a)

BIT-REVERSE(a, A)
n := length[A]
for s := 1 to lg n
     do m := 2^s
     wm := e2πi/m
     w := 1
     for j := 0 to m/2 - 1
          do for k := j to n - 1 by m
               do t := w . A[k + m/2]
                    u := A[k]
                    A[k] := u + t
                    A[k + m/2] := u - t
               w := w . wm
return A
```

3.3.2 NuFloW Code

The top level view of the NuFloW code is shown below:

[1]Cormen Thomas, Leiserson Charles, Rivest Ronald , "Introduction to Algorithms", MIT Press 1991, p794

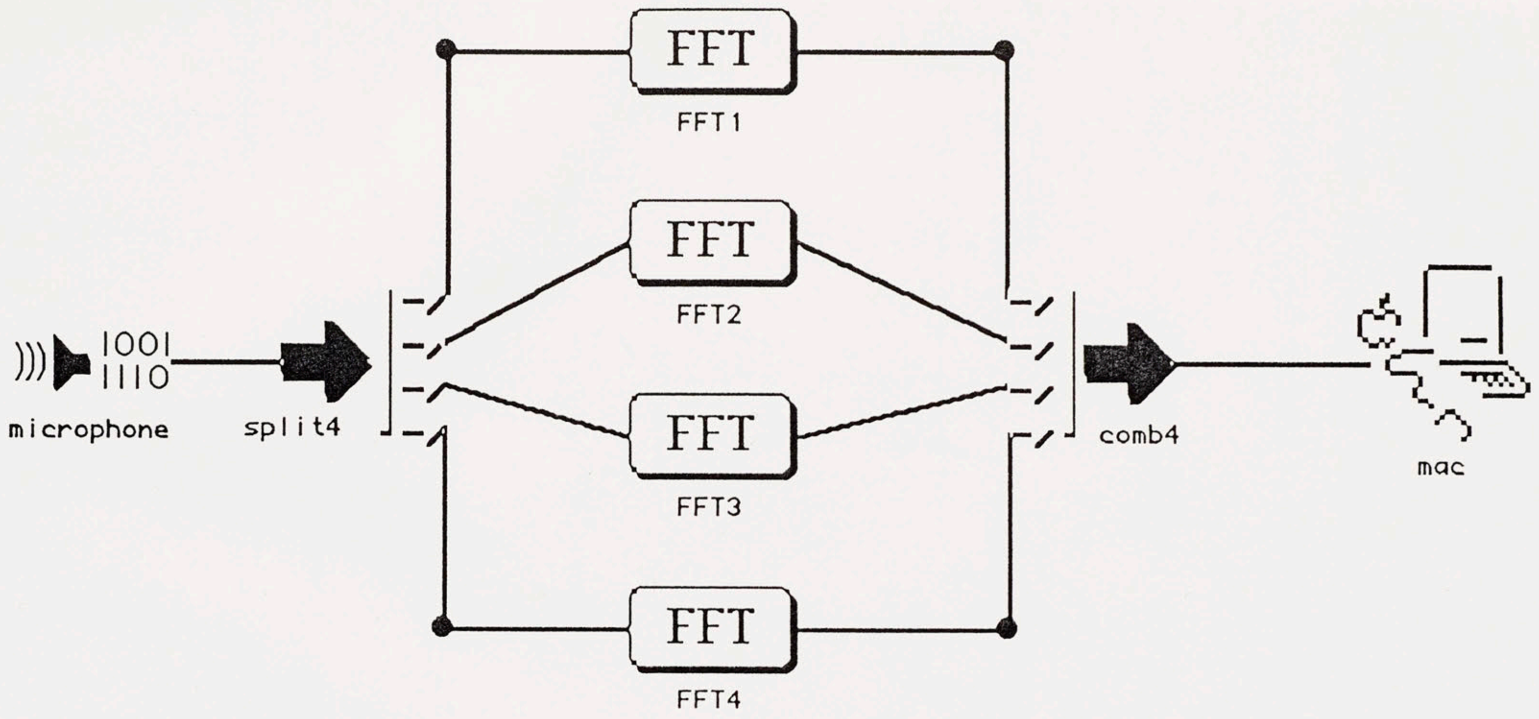


Figure 3.3: Top-level view of a four node Spectogram

The code is designed to run on 5 NuMesh nodes, 4 DSP nodes computing FFTs and 1 node serving as the I/O board. The splitter module takes a stream of data from the I/O board and splits it in a sliding window fashion, to the four FFT modules. The output of the four FFT modules is combined and send to the Macintosh, which is used to display the spectrogram.

Each FFT module shares the same code definition file which is shown below:

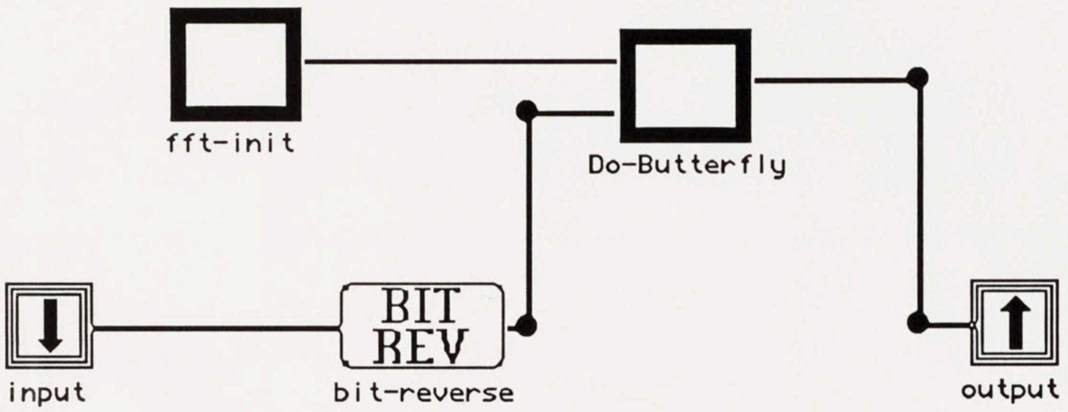


Figure 3.4: FFT module

In the first step, the array of inputs is bit-reversed, and some initial values are computed by the fft-init module. Note that the length of the input array is fixed, which ensures predictable communication patterns. The Do-Butterfly block corresponds to the nested do-loops of the pseudo-code and executes the butterfly:

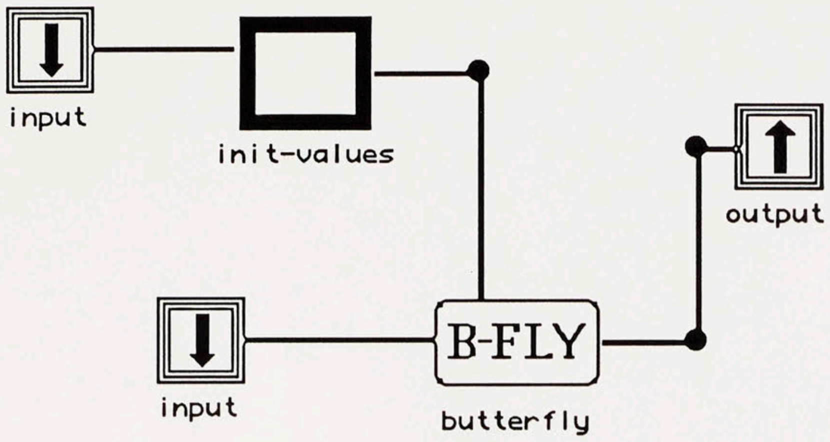


Figure 3.5: Butterfly module

The init-values block provides initial values to the butterfly block, which performs a standard butterfly operation on the input array. Input and output blocks can have the same name, since they are not included in the NIFT file.

3.3.3 Source Code

The definition of leaf blocks are written in C. These files are not appended to the NIFT output for 2 reasons: 1) A file might include other files as header definitions etc.. which would also need to be appended. 2) NuFloW is designed to edit graphic files, and provides a minimum of text editing functions. It is hoped that NuFloW will be integrated to emacs, when it is ported to UNIX. In fact, code definition files of source level blocks should reference the make file for that block. In this way, the compiler could follow the directives specified in the make file.

3.4 NIFT output

A partial NIFT listing for the spectrogram example is included below:

```
(header
   (paradigm "stream")
   (NIFT-version 0.1)
   (date "8/29/91")
   (blocks 20)
   (creator "PPL"))

(block
   (name @root-block@)                        ;;; root block
   (ports)                                    ;;; no port + fixed name
   (sublocks
      ("microphone" "microphone")
      ("split4" "split4")
      ("fft1" "fft")                   ;;; all ffts reference same block
      ("fft2" "fft")
      ("fft3" "fft")
      ("fft4" "fft")
      ("comb4" "comb4")
      ("mac" "mac"))
   (streams                            ;;; only local streams are listed
      (("microphone" 1) ("split4" 1))
      (("split4" 2) ("fft1" 1))
      (("split4" 3) ("fft2" 1))
      (("split4" 4) ("fft3" 1))
      (("split4" 5) ("fft4" 1))
      (("fft1" 2) ("comb4" 1))
      ..
      (("comb4" 2) ("mac" 1)))

(block
   (name "microphone")
   (ports (wo))
   (source))                ;;; no source code

(block
   (name "split4")
   (ports (ro) (wo) (wo) (wo) (wo))
   (source "split4.cfsm"))

(block
   (name "fft")
   (ports
      (ro ("bit-reverse" 1))           ;;; matching of ports between
      (wo ("out1" 1)))                 ;;; parent and child
      (sub-blocks
         ("fft-init" "fft-init")
         ("bit-reverse" "bit-reverse")
         ("do-butterfly" "do-butterfly")
      (streams
         (("bit-reverse" 2) ("do-butterfly" 1))
         (("fft-init" 1) ("do-butterfly" 2))
         ...)

   (block
      (name "bit-reverse")
      (ports
         (ro)
         (wo))
      (source "bit-reverse.c"))


   ...
```

NIFT files could become more compact by replacing all strings (except for filenames) with numbers. Strings are only included for debugging purposes.

Chapter 4

Making it all Work!

4.1 Intoduction

The NuFloW programming environment is just one of the many components of the NuMesh front-end. The last chapter showed that it was possible to create applications using NuFloW. A lot of work still needs to be done though, before NuFloW becomes fully operational. The NuFloW interface lets the user assemble tasks together and produce a textual representation of a project. But compilers are needed to generate binary code for the source language blocks, and a scheduler needs to be designed to support the stream paradigm. Section 4.2 and 4.3 discuss some of the changes which the run-time and compile-time environment of the NuMesh must undergo. Section 4.4 offers some ideas for future work while section 4.5 concludes.

4.2 Run-Time Environment

A run-time support program, (ambiguously) called NuMesh, is used to load programs onto the NuMesh hardware, and service the mesh's I/O requests. The NuMesh program senses a particular NuMesh configuration, which is then displayed on the screen. Each NuMesh node is depicted by a particular icon. The user is able to specify both a CFSM and PE binary file to be loaded on a node. A natural extension to NuFloW would be to interface it with the NuMesh loader program. A user could drag a NuFloW block to the NuMesh topology window, thereby assigning the computation of that task. In fact, the topology window would become the root block, which is synthesized during the compilation of NIFT files. At first, only top-level blocks would be dragged onto the topology window. But as the compiler and scheduler technology get better, the user should be able to assign every NuFloW block to a node.

The distinction between CFSM and processing element code is eliminated. Tasks are dragged onto NuMesh nodes to indicate where they should execute, and the compiler figures out which task should be serviced by the CFSM or processing element part of the node, or both. The user should not think of CFSM code as being any different from processing element code. In fact, when a user connects two blocks with a stream, some CFSM code is automatically generated to support the `read_port` and `write_port` operations between the blocks. Because the compiler and scheduler manage the allocation of tasks on the mesh, the user does not need to specify any "glue" CFSM code.

4.3 Compile-Time Environment

The current compiler and scheduler technology is inadequate for the NuFloW programming paradigm. In the short-run, it may be possible to modify the scheduler to support NIFT. In the long-run though, a new compiler and scheduler will need to be designed and implemented. The compiler must be able to provide some timing information to the scheduler. A new language, Bil [3], may be used as an intermediary, hardware independent language to provide timing estimates. Bil is designed to provide very accurate upper-bound estimates on execution times and is well suited to the NuFloW environment. Indeed, it assumes that code is written in blocks, which are then mapped to NuMesh nodes. The scheduler is the heart of the NuMesh system. It allocates tasks among the various processors and programs the CFSMs to support their communications. The following two sections describe tools aimed at facilitating the interactions of the compiler and scheduler with the NuFloW environment.

4.3.1 Connectivity Checker

Type-checking dramatically reduces development time. NuFloW provides mechanisms to perform some type-checking for node blocks: If a stream connects two node blocks, NuFloW will check that both ports have compatible read-write privileges. Furthermore, if the user has specified additional type information (see section 2.3.2), NuFloW will check that the types match. NuFloW is not able to check typing information for leaf blocks, since it cannot parse source languages. A NuFloW tool could assist the compiler in the initial stage of compilation. Such a tool would read a NIFT file, parse the leaf blocks' text files, and check the typing information. This connectivity checker[1] is invoked by the compiler, but should be implemented as a stand-alone module. This tool is language specific, and checks that:

• every `write_port` is balanced by a `read_port` and vice versa.

• all ports all valid and referenced.

• the read/write privilege of every port is not violated

• if the user has specified additional typing information, the typing is matched.

4.3.2 Block manipulators

NuFloW does not address the issue of block granularity. A leaf block could contain a few lines of code as well as a few thousand lines. In the short term, the scheduler will require the NuFloW user to indicate where each block should run. In the long run, however, the scheduler will allocate blocks of code on the Mesh. Because the scheduler must accommodate blocks of different granularity, it will need to split, combine, and move blocks within a project to balance the computational load of each node. The splitter and combiner work to decrease/increase the computational demands of a block. The mover is called by the scheduler to move a block to a new hierarchical position. These tools will be called by the scheduler.

[1]The pre-compiler tools were proposed by Chris Metcalf in: "NIFT...", NuMesh Memo #8.

4.4 Future Work

The last two sections focused on ways to interface NuFloW with the rest of the NuMesh software. The following section proposes some changes to improve the current implementation of NuFloW.

4.4.1 Compile-time parameters

In NuFloW, all blocks which share the same code definition perform identical tasks. There is no mechanism to let blocks share the same code definition, save a number of modifications. It would be very useful to allow blocks to be parameterized. For instance, in the spectrogram program, the node connected to the I/O board may perform a slightly different computation than the rest of the nodes. Thus, the user should be able to declare labels at the user interface label that the compiler could then test for. Generalizing the idea of labels, it should be possible to specify a set of compile time constants. Changing a compile time constant would change the behavior of the program once it is recompiled and executed.

Another drawback with the current implementation of NuFloW is that although NuFloW abstracts out the communication details, it makes assumption on the number of NuMesh nodes that will be used to perform the computation. In the spectrogram example included in Chapter Three, the four FFT blocks imply that four (or less) DSP nodes will be used. The four nodes could be arranged in any topology: square, linear of T shaped. But to run the spectrogram on five or more nodes, the user would have to create a new NuFloW program containing five or more instances of the FFT block. A

NuFloW program should be able to abstract out the number of processors used as well as the topology of the NuMesh configuration. The spectrogram code should depend on a parameter specifying the number of processors required:

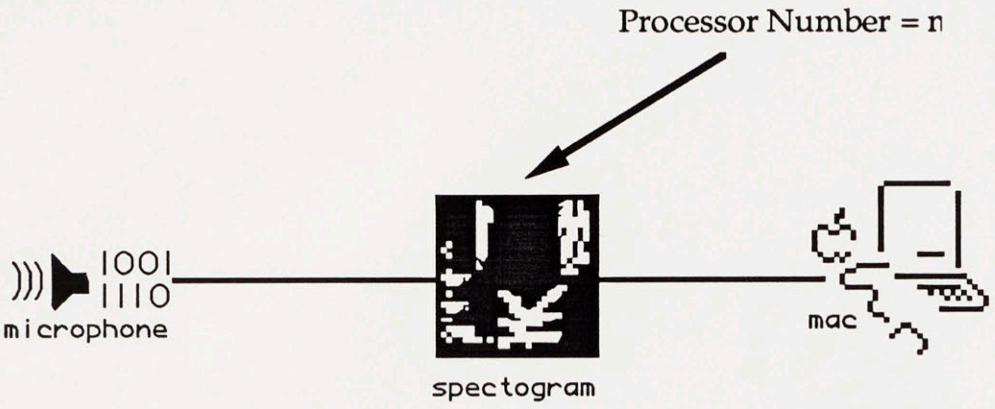


figure 4.1: Processor independent spectrogram demo.

Given a number n of processors, the NuFloW editor should be able to produce a NIFT file containing n instances of the same code. The difficulty with replicating blocks automatically stems from the fact that a scheme must be used to formally describe how stream connections within the parameterized block scale with the replicating factor. Specifying the stream connections for the spectrogram block is straightforward. Each of the n-inputs of the splitter goes to an instance of the FFT block. The output of each FFT block goes to one of the n-inputs of the combiner. In other cases, however, specifying stream connections may not be as simple. Because the spectrogram uses a sliding window algorithm to feed the FFT blocks with data, parameterizing a combiner or splitter block is difficult. The programmer must specify how each of the input and output stream connect to the blocks. Thus to successfully parameterize blocks, research will need to be conducted to design a formal textual language that permits the programmer to specify stream connections algorithmically. Even if the language closely resembles C, an interpreter will be needed to execute the replicating code.

4.4.2 Run-time parameters

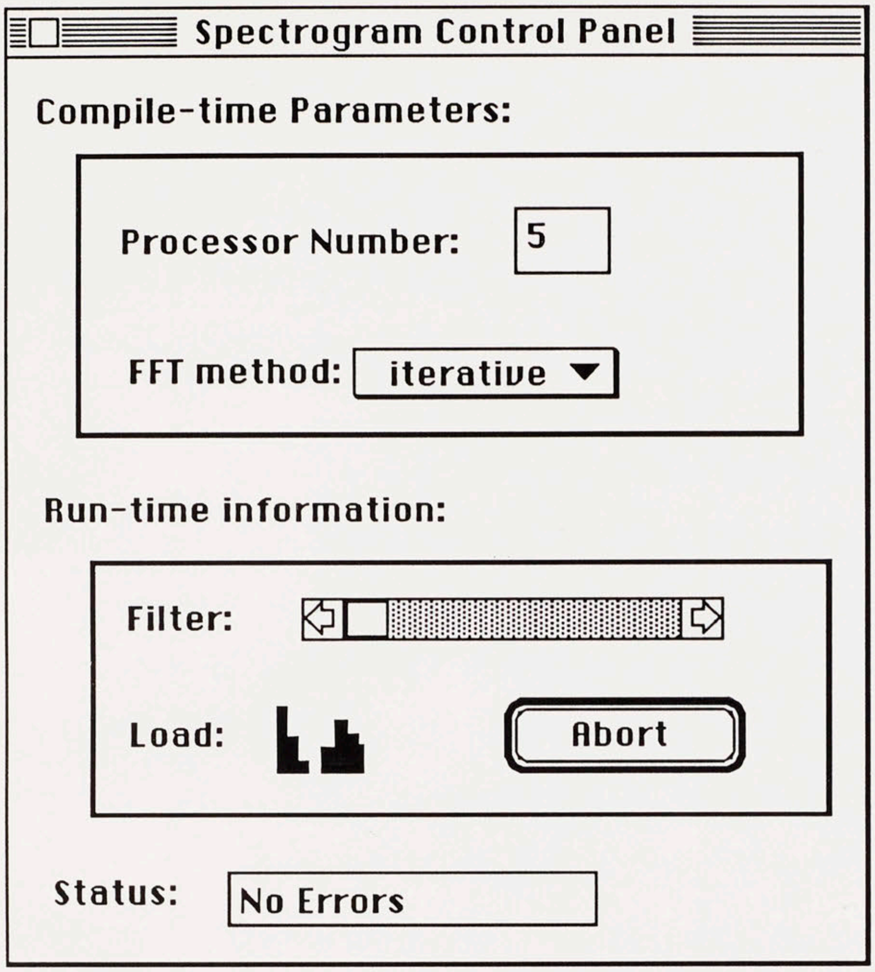
So far, we have discussed the need for various kinds of compile time parameters. A block should also contain a set of run-time parameters. Run-time parameters differ from compile-time parameters in that they can affect the course of a computation in its progress. At the compiling stage, variables which might be changed by the user during the computation should be declared specially. When the compiler encounters such a variable, it could include some glue code that could set the variable to its new value. A protocol would also need to be designed to let the NuMesh host send data to a particular node at specified intervals. The scheduler could, for example, reserve a number of cycles every so often to let NuMesh nodes read messages from the host, and process them. Each message would contain a change/ignore order, possibly followed by the identification number and value of the variable to modify. Of course, the user should be able to specify the frequency at which nodes check to see if they must modify these pre-defined run-time parameters.

4.4.3 Reporting messages

The above protocol can be augmented to permit communications from the nodes to the NuMesh host. At regular pre-defined intervals, nodes could send status information back. The host could then alert the user if a node encountered a system or user-defined error. Other types of messages could be defined to report the state of a computation. Note that applications that make intensive use of the host's I/O cannot use this protocol for all communications, as it is assumed that this reporting mechanism will be active for a small fraction of NuMesh cycles.

4.4.4 Control Panel Toolkit

To support the above protocol, the user must have access, at the user interface level, to a standard set of tools that will permit him to easily interact with the appliaction running on the NuMesh. A control panel toolkit could be included with NuFloW, which would enable the user to assemble in a window various controls to send and receive information from the mesh. Each block could be linked to a panel, which would control the various compile-time and run-time parameters.


Spectrogram Control Panel

Compile-time Parameters:

Processor Number: 5

FFT method: iterative

Run-time information:

Filter:

Load:

Status: No Errors

Figure 4.2: Spectrogram control panel

4.5 Conclusions

NuFloW was designed to serve as a complete programming environment for the NuMesh computer. First, NuFloW needed to assist the programmer writing code for the NuMesh. The current implementation of NuFloW achieves this goal by supporting a stream based programming paradigm using a pictorial language. The pictorial language represents tasks by graphical blocks, and lets blocks communicate through streams. The stream based paradigm was chosen because it mirrors the range of communications that can be supported by the NuMesh static routing policy. In addition, using blocks and streams abstracts out communication details from the user, who is not required to understand the details of the NuMesh hardware.

A second goal of NuFloW was that it should allow the NuMesh to be programmed using any textual language. Currently, NuFloW does not make any assumptions on the source language used. In fact, blocks within a same project could be written with a combination of languages. Using textual languages in addition to blocks and streams is a key attribute of NuFloW. Text is an efficient and common medium to perform actual calculations. Providing all primitive operators graphically would have resulted in an incomplete and inefficient language.

Finally, NuFloW was to serve as a central user interface to the various components of the NuMesh software. A specific file format, NIFT, was designed to facilitate the compilation of NuFloW programs into executable NuMesh binaries. NIFT is compact, flexible, and disassociates source text files from the graphical blocks. Thus, the user is able to connect blocks and streams graphically, but work on each source (textual) block completely independently.

References

[1] Agarwal, Chailken, Johnson, Kranz, Kubiatowicz, Kurihara, Lim, Maa and Nussbaum: *"The MIT Alewife Machine: A Large Scale Distributed-Memory Multiprocessor."* To appear in "Scalable Shared Memory Multiprocessors", Kluwer Academic Publishers, 1991

[2] Crowther, Goddhue, Starr, Milliken and Blackadar: *"Performance Measurements on a 128-Node Butterfly Parallel Processor."* BBN Laboratories.

[3] Fetterman, Michael: *"BIL: an Intermidiate Language for NuMesh."* Master's thesis. MIT Department of Electrical Engineering and Computer Science, January 1991.

[4] Hillis, Daniel: *"The Connection Machine."* Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, June 1986.

[5] Honoré, Frank: *"Redesign of a Prototype NuMesh Module"*. Bachelor's Thesis, MIT Department of Electrical Engineering and Computer Science, May 1991.

[6] Laffont, Philippe: *"NuMesh: a Software Overview."* Internal Document.

[7] Nguyen, John: *"CFSM Assembler Description."* NuMesh Memo #2.

[8] Nguyen, John: *"A C Interface for NuMesh."* NuMesh Memo #3.

[9] Nguyen and Pratt: *"Synchronization of Hardware Oscillators in a Mesh-Connected Parallel Processor"*. Forthcoming.

[10] NuBus Data Book Products. Texas Instruments, 1990

[11] Metcalf, Christopher: *"NIFT: NuMesh Interchange Format for Text."* NuMesh Memo # 8.

[12] Metcalf, Christopher: *"Running Multigrid on a NuMesh."* NuMesh Memo # 5.

[13] Peterson, Sutton and Wiley: *"iWarp: A 100-MOPS, LIW Microprocessor for Multicomputers."* IEEE Micro, June 1991.

[14] Pezaris, John: *"CFSM Revision 2: Progress to Date"*. Numesh Hardware Memo #9 & #12, Spring 1991

[15] Seitz, Charles: "The Cosmic Cube." Communications of the ACM, Volume 28, Number 1. January 1985

[16] Tessier, Russel: *"Sparc Based Processing Element / LPI Redesign"*. NuMesh Hardware Memo #10, May 1991

[17] Trowbridge, Sean: *"A Programming Environment for the NuMesh Computer."* Master's thesis. MIT Department of Electrical Engineering and Computer Science, May 1990.

[18] Ward, Stephen: *"NuMesh, a Scalable, Modular, 3D Interconnect."* MIT Laboratory for Computer Science's Computer Architecture Group. Internal document. February 89.

[19] Ward, Stephen: *"Towards LegoFlops, RecognizingSpace in the Digital Abstraction."* MIT Laboratory for Computer Science's Computer Architecture Group." Internal document. January 1991.