A Formal Model of Parallel Computation for LGDF2

David C. DiNucci

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-006

December, 1988

A Formal Model of Parallel Computation for LGDF2

David C. DiNucci

Department of Computer Science and Engineering
Oregon Graduate Center
19600 NW Von Neumann Drive
Beaverton, OR 97006 USA
(503) 645-1121

Abstract — **LGDF2 is a new and growing language for expressing portable, efficient parallel programs. In this paper, the computational model of the language, the F-net, is presented formally in terms of a virtual machine. Precise methods of representing and reasoning about computations are defined. The model facilitates the design and implementation of architecture-independent parallel programs and gives the designer control over non-determinism. Restrictions imposed by the model are clarified, and applicable work by others is identified.**

Index Terms — **Large-Grain Data Flow 2, Parallel Programming, Graphical Programming, Virtual Machine, F-Nets, Non-determinism, Architecture Independence, Parallel Program Design, High-level Parallel Programming.**

1. INTRODUCTION

The Large-Grain Data Flow (LGDF) language has evolved over the years as an attempt to make parallel programming more palatable. It began as a software engineering experiment—adding execution semantics to the data flow diagrams used in structured analysis methodologies[1]. It became apparent that the semantics could be implemented efficiently on shared memory multiprocessors, and this was successfully demonstrated with a scheduler for the HEP 1 [2].

In an attempt to formalize the syntax and semantics of the language, to allow more efficient implementation on shared memory processors, and to enable implementation on distributed memory architectures, the language has recently undergone another transformation. The principles of this new version, LGDF2, have been demonstrated with an efficient implementation on a shared-memory parallel processor, the Sequent Balance [3]. However, the language is still new, and much has yet to be learned about its power, techniques of using it effectively, and exactly what restrictions must be placed on its semantics to make it truly portable. This paper will address some of these issues by separating the computational model of the language from the language syntax and features.

Like all high-level computer languages, LGDF2 can be considered on three levels:

(1) A definition of the "basic" virtual machine that it presents to the user. The valid statements in the language define the instruction set of the machine, and the basic data types define the properties of various kinds of memory.

(2) The abstractions that are built into the language itself, but are easily defined in terms of the basic virtual machine. In sequential languages, structured control statements (if, case, for, while) fit into this category. The use of these slightly higher-level abstractions can make the intention of the program clearer, to both the reader and the language processor, than if the base equivalents in the virtual machine are used.

(3) A set of meta operations that allow the user to build higher level abstractions—for example, `typedef` and function definitions in C.

This paper discusses the first two of these levels of LGDF2 and describes some ways of reasoning about them. Section 2 briefly describes the first principals of MIMD architectures. Section 3 lists the goals of LGDF2, and Section 4 reviews some other work that addresses some of those goals. Section 5 presents the syntax and semantics of an architecture-independent virtual machine (VM) for executing LGDF2 programs. Section 6 explains why a better concept of "execution" is needed, and offers one in the form of partial orderings of events called Execution Graphs. Section 7 provides motivation for some of the semantics of the VM, and Section 8 illustrates that some straightforward extensions to the VM can be implemented in terms of the basic VM instructions. Section 9 relates the VM to other computational models, and section 10 contains a summary and conclusions.

2. Overview of MIMD Computer Architecture

At the heart of almost all general-purpose Multiple-Instruction Multiple-Data (MIMD) processors is a set of standard sequential CPUs. One of the main differences in the classes of these machines lie in the way the address space is allicated to these individual CPUs. In shared memory systems, all memory is addressable by all of the CPUs. In distributed (or private) memory systems, no two CPUs have access to the same memory. In hybrid systems, each CPU has some private memory while other memory is shared. In any system, shared memory can be divided into local (fast) and remote (slow) address spaces.

Since any processor can store data into shared memory, and any other processor can access that data, shared memory provides a high-bandwidth (memory speed) medium for communication. Shared memory programming is primarily concerned with ensuring that the individual processers access the shared memory in a coordinated and predictable fashion. To accomplish this, some form of "meta-communication" must be performed so that each processor can know when it is its "turn" to safely access some portion of shared memory. This meta-communication is usually based on the notion of a semaphore (or lock). Considered as an abstract data type, a lock has two operations which can be performed on it:

- Unlock the lock.
- Wait until the lock is unlocked, then lock it.

Though protocols exist for building an abstract data type with these semantics using only shared memory and the standard `load`, `store`, and `test` instructions that exist on the component processors, special hardware instructions and/or "atomic lock memory" is usually provided in order to make lock operations more efficient. Additional lock support (e.g. waiting for a lock to be locked before unlocking) is sometimes provided in hardware or software, but the operations described here are sufficient to implement them.

All distributed memory systems, and some hybrid systems, provide another form of communication called message passing. In it, a process can instruct special hardware called a channel to make a copy of some contiguous portion of its address space and send it to another processor in the form of a message. Some architectures also support a broadcast facility, where multiple processors can be named as destinations. When a processor notices an arriving message (either by polling or by being interrupted), it stores the message into a buffer area until it is requested by a process running on it. At that time, the message is either copied into a buffer specified by the user process or the process is simply informed of the address in the buffer area where the message is stored. Once a message is delivered to a requesting process, it will not be delivered to another.

If a process requests a message before one arrives, this fact is recorded by the message reception mechanism (software or firmware) so that the next message to arrive can be immediately forwarded to the requesting process. The process may have the choice of blocking until the message arrives, or it may continue processing and poll

the message reception mechanism periodically.

Another model of communication, sometimes called "generative communication", is now being supported directly by some architectures[4]. Based on the Linda language, it views all communication as depositing or withdrawing information from a large relational-database-like structure called "tuple space" (or "object space"). Rather than give a specific location or processor as a source or destination for messages, actions on the tuple space refer only to characteristics of the tuple being deposited or withdrawn from the space. Even in this model, however, the underlying communication takes place through message-passing or shared-memory, though it may not be directly visible to the user.

3. Goals and Objectives of LGDF2

Many of the goals outlined in this section became goals only after it became clear that they might become achievable. They can be seen as subparts of one overall goal: to facilitate the construction of efficient, portable parallel programs capitalizing on as much existing technology as possible.

3.1. Portability of Parallel Programs

A major goal of the virtual machine is portability among general-purpose MIMD processors. If the term portability is taken to mean the ability to run on any general purpose MIMD processor given the proper software run-time support, then nearly any message-passing model is portable since the message-passing semantics can be implemented by copying to/from shared memory. In addition, portability should encompass the idea that its implementation on a processor would use the same constructs for achieving efficiency as a programmer would if he/she were writing the program specifically for that architecture. Portability also implies independence of the speed and number of processors.

3.2. Accept Implementation Advice

Though a program defines a precise semantics for execution, it does necessarily exactly specify how that semantics will be achieved. Because of the very different qualities of communication on different architectures, our portable model will have to rely heavily on this declarative aspect of programming. Even on a given architecture, there are often many ways of implementing the same program. The preferred way may be dependent on factors, such as the frequency of a particular operation or the amount of contention for a resource, which may not be apparent or derivable from the program syntax.

Programmers tend to have higher level knowledge about the resources needed by a program, and should be able to advise the implementation on these matters. This may include (but need not be limited to) suggestions on

- assigning processes to processors
- data communication methods

- scheduling priorities
- use of fast and slow memory

This advice need not be part of the model, however, since its presence or absence will not (must not!) affect the program semantics and will be likely be very specific to the target architecture.

3.3. Use Existing Compiler Technology

The heart of a general-purpose MIMD computer is a set of general-purpose sequential processors. Computer scientists have spent decades designing languages and compilers for these machines, and have developed a wide variety of techniques and paradigms to use them effectively. Any approach to parallel processing that begins by avoiding the use of these languages is "throwing the baby out with the bath-water". It shouldn't be necessary to compromise our ability to get good efficiency from the component processors in order to use more of them.

3.4. Preserve Sequential Modular Reasoning

MIMD parallel programs today *are* typically written in traditional languages, complemented with subroutine calls, macro calls, or compiler extensions to give the user access to the parallel hardware of the machine. These new primitives may be very low-level, like a lock or message send, or higher-level, like a monitor [5] or access to shared "tuple space" [4], but no matter how high-level they seem, they all have a major drawback: they deprive us of our ability to reason about the code in a modular fashion. *The correctness of the code suddenly depends on the correctness of other code which is not within the module. In fact, it depends on* all *the other code, since any other code can cause errors by an incorrect message send, deposit to tuple space, or write to shared memory.*

This is the most serious criticism of these approaches, since abstraction is at the core of our ability to build large programs, whether sequential or parallel. If a module's implementation is not hidden, we cannot reason about it in terms of its specification: the complexity of its specific implementation must be carried and dealt with throughout its lifetime.

3.5. Increase Visibility of Parallel Interactions

Parallel primitives embedded in sequential code not only make sequential reasoning hard or impossible, they also hide the parallel structure of the program. There is no central location to find which processes interact and which don't, or even more useful, to dictate which processes should interact and which shouldn't.

3.6. Allow Non-deterministic Execution

Nondeterminism is often considered an undesirable property of computation. Stating that a program is non-deterministic is often regarded as evidence that it does not work correctly. Indeed, unwanted nondeterminism can often sneak into parallel computations without the programmer's knowledge.

In fact, many non-numerical applications require, and many numerical applications benefit from, a first-come first-served style of programming, resulting in behavior which is not completely determinable by the input alone. The programmer should be able to reason with and control such "useful" non-determinism, rather than fear it. This means that it must be apparent to the programmer when it exists, and deciding which actions will eligible for non-deterministic execution must be completely under the programmer's control.

4. Related Work

To the author's knowledge, no other model has attempted to simultaneously satisfy all of these goals. The portability goal is seen as unimportant to many, since there are only two principle classes of architecture (shared and private memory), and "relatively portable" paradigms (such as monitors and message passing, respectively) can be used to program the classes individually[6]. Even with this approach, however, most of the other goals go unaddressed.

Declarative programming models are often espoused as offering solutions to many of these goals. These include functional, dataflow [7] [8], and logic languages. The premise is that the order of evaluation (i.e. execution) in these languages is dictated solely by the data dependencies that exist among the otherwise independent functions. Therefore, if multiple functions are ready to evaluate and have no data dependencies among them, they are candidates to evaluate concurrently on separate processors.

This concept is very valuable, and the model described in this paper will make use of it. The existing declarative languages fail some of our goals, however.

- We must adopt a new paradigm for all programming in order to use them, rather than using existing sequential languages for programming the sequential processors.
- With the vast amount of potential, fine-grained parallelism available, partitioning the problem (dependency graph) among processors must be automated; it is unrealistic to assume that the programmer could have much control over implementation, or could even understand how the compiler chose to implement the program.
- Non-deterministic execution is directly contrary to the functional approach (though, to be fair, it is not difficult to augment a functional language with non-deterministic non-functions in practice).
- The languages are based on a zero- or single-assignment notion, giving the programmer little power to specify efficient use of memory or to minimize copying.

In all, such declarative languages require that all sequential and parallel implementation issues be left up to the compiler, even if efficient methods of implementation are obvious to the programmer. The drawbacks mentioned will be obviated only by the advancement of new programming techniques and compilers that will consistently implement declarative programs as well or better than a programmer with more control. This is not likely to occur soon, if ever.

5. The MIMD Virtual Machine

5.1. Introduction

The computational model of LGDF2 will be described by defining the virtual machine that it presents to users, in part to separate the features of the machine from the syntax of LGDF2. The programs for the machine will be specified both textually and graphically. Though the graphical form is similar in many ways to the data flow graph associated with LGDF2 syntax, it will be described as a separate entity called an F-Net to avoid this (possibly incorrect) correspondence.

This MIMD VM is different than most, in that the instruction set is not fixed, but rather, implemented with easily-replaceable microcode. The intention is that the programmer tailor the instruction set to each particular application. An instruction execution on this machine would be more typically considered as the execution of a process (named by the opcode).

5.2. Syntax

5.2.1. Hardware

The VM consists of *temporary memory*, *input memory*, *output memory*, and *instruction memory*. Instruction memory is read-only. All memory other than instruction memory is referred to collectively as *data memory*, and a single data memory location is referred to as a *state*. There are no registers (all operations are memory-to-memory), and, unlike a typical sequential VM, there is no instruction counter.

Each data memory location (i.e. state) is capable of storing a data structure of arbitrary size, called its *data state*, and an additional value called its *control state* which can assume the values **Left**, **Right**, or **Neutral**.

5.2.2. Software

Each instruction memory location is capable of storing one *instruction* of the form

<opcode> *<operands>*

where *<operands>* is a list of one or more data memory references, each of the form:

<input-memory-address>`,Right`

or

<output-memory-address>`,Left`

or

<temporary-memory-address>`,`*<side>*

where *<side>* is `Left` or `Right`.

5.2.3. Firmware

Each operation in the instruction set of the VM takes a fixed number of operands, and defines a functional mapping from the data states of some of them (the *readable* operands) to new data states for some of them (the *writable* operands) and *transitions*

for all of them. The readable and writable operands need not be mutually exclusive. A transition is one of **reserve, grant,** or **neutralize.**[1]

The mapping is intended to be expressed in the form of a program, called *microcode.* This may be written in a standard sequential higher-level language, though any language capable of expressing the mapping (including functional or parallel languages) will do. The microcode sees the data states of the operands as call-by-address arguments, and the mapping of readable data states to writable data states is expressed by the use and/or modification of these arguments. The microcode language is augmented with primitives to effect **reserve** and **grant** transitions for an operand. The only restriction on the microcode (other than its functional nature) is that it not reference an operand in any way after it has prescribed a transition for it using one of these primitives.

Each opcode has a permission signature that summarizes the role of each of its operands in the mapping: whether it is readable and/or writable, and whether a **grant** and/or **reserve** transition is ever prescribed for it. The permission signature for an n-ary operation is denoted by an n-tuple of sets, each containing one or more of the codes `rd`, `wr`, `gr`, or `rs`. The i-th set corresponds to the i-th operand.

The **neutralize** transition is not explicitly mentioned in the permission signature, nor is it explicitly prescribed by the microcode. It will become clear later that the **Neutral** control state and the **neutralize** transition is present only to guarantee a representation for microcode which encodes a partial function, and that the **neutralize** transition will only be required (or desirable) for operands with the permission signatures containing either **wr** or the combination of **gr** and **rs.**

5.2.4. F-net

The instructions and permission signature for a program for this machine may be represented graphically as an *F-net.* In it, each state is represented as a square box labeled with its address. Input memories are distinguished by an arrow pointing at their left side, output memories by an arrow pointing from their right side. Each instruction is represented as a circle labeled with its opcode, connected with arcs to the boxes representing its operands. Each arc is numbered to denote which operand it represents, to preserve information on the order of the operands in the net. The arc is connected to either the left or right side of a box, depending on the <*side*> field of the operand.

Each arc is then annotated to denote the permission signature of the opcode/operand that it represents as follows:

(1) `rd` - An arrowhead toward the instruction.

---

[1] One exception: If a **neutralize** transition is associated with a writable operand, the new data state of the operand is unimportant, and need not be functionally determined by the data states of the readable operands.

(2) `wr` - An arrowhead toward the state box.

(3) `gr` - A continuation of the arc through the side of the state box. (Meant to suggest a finger pointing to the opposite side of the state box.)

(4) `rs` - A semicircle within the state box, beginning at the end of the arc. (Meant to suggest a finger pointing to the same side of the state box.)

See Figure 1a for an example of an F-net, and Figure 1b for a contrived example of possible microcode associated with it, expressed in a C-like pseudo code.

5.3. Semantics

Configuration

A *configuration* consists of an F-net, an *input sequence* of data values for each of

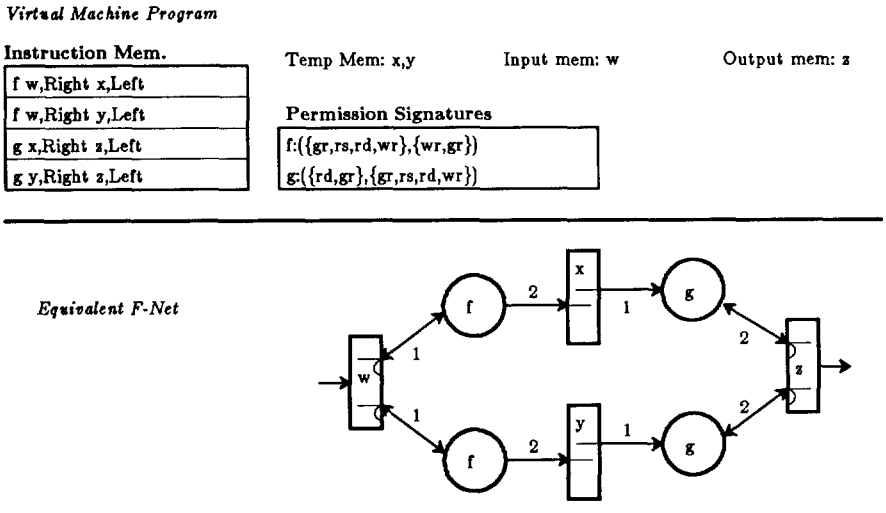

Figure 1a. Sample VM program and associated F-Net.

```
define_op f(next_n,result)      | * Cube n and decrement it *
   n = next_n;                  | Take copy of number to cube
   next_n = next_n - 1;         | Decrement it for the next guy
   if (next_n == 0)             | If this was last n
      grant next_n;             |     change control state of n
   else
      reserve next_n;
   result.n_cubed = n*n*n;      | Compute result = n cubed
   result.last = (n==1);        | Tell whether this is last result
   grant result;                | Finished with result
-------------------------------------------------------------------
define_op g(term,term_sum)      | * Add term to sum *
   term_sum += term.n_cubed;    | Add term to sum
   if (term.last)               | If this is last one
      grant term_sum;           |    change control state of result
   else
      reserve term_sum;
   grant term;                  | Finished with term
```

Figure 1b. Microcode for F-net in Figure 1a.

its input memories, an *output sequence* of data values associated with each of its output memories, and a *state value* of the form (data state, control state) associated with each state. An input (output) memory with its associated input (output) sequence will be referred to together as an *input* (*output*). An *initial configuration* is a configuration for which all output sequences are null (have length zero) and all state values are of the form (**ZERO**,Left).

Enabled

If the control state portion of a state is Left (Right), the left (right) side of its box is said to be *active.* An instruction is *enabled* if it is connected only to active sides of boxes: i.e. the control state of the memory location referenced by each of its operands is equal to the <*side*> field of the operand. An input is *enabled* if it has a non-empty input sequence and the input memory has a control state of **Left.** An output is *enabled* if its output memory has a control state of **Right** and a data state other than **EOF** (a distinguished value for output memories).

Execution Step

An *execution step* consists of performing any enabled instruction, input, or output.

- An instruction is *performed* by evaluating the function associated with its opcode (typically by executing the microcode) with the readable operands specified in its operand fields. The data states of the writable operands are replaced with the results of the evaluation, and the control states of all of the operands are transformed according to the transitions resulting from the evaluation as follows:

  reserve - the control state of the operand is left unaltered

  grant - the control state is changed from Left to Right or Right to Left

  neutralize - the control state is changed to Neutral

- An input is *performed* by setting the data state of the input memory to be equal to the first element of the input sequence, deleting the first element of the input sequence, and setting the control state of the input memory to **Right.**
- An output is *performed* by appending the value of the data state of the output memory to the end of the associated output sequence and setting the control state of the output memory to **Left.**

Execution Sequence

An *execution sequence* consists of a sequence of execution steps on an initial configuration.

Final Configuration

An F-net is said to have entered a *final configuration* if all output memories have a state value pair of (**EOF**,Right). Note that execution may be able to continue even after the F-net has entered a final configuration, but in no case will it leave a final configuration or produce more output.

5.4. Discussion

5.4.1. Parallelism

The semantics make no mention of parallel execution. If they did, it could be hard to determine which aspects of an instruction's execution relied on other concurrently executing instructions. As is, it should be apparent that instructions can be reasoned about independently, and that deadlock is not possible between instructions.

It should also be apparent that much of the execution could proceed in parallel with no ill effects. For example, two enabled instructions with mutually exclusive operands can be executed in either order or concurrently, and the outcome will be identical, as long as the microcode does not share temporary variables with other microcode (an assumption we will make and enforce).

5.4.2. Non-determinism

An F-net which has at most one instruction connected to each side of each state must be deterministic, since every step in the execution in functional. This is a special case of the same property shown for any network of functional processes connected by fixed channels (a Kahn-MacQueen network) [9]. Non-determinism can be introduced into an F-net in only one way: by connecting multiple instructions to the same side of a single state such that there is some execution sequence which can cause more than one to be enabled at the same time. Thus, non-determinism can be present only between instructions (within the F-Net) rather than within the instruction's microcode.

5.4.3. Powerful Microcode

If it is assumed that an implementation has the resources to execute any number of the enabled instructions in an F-net concurrently, either by putting them onto separate processors or by timesharing, the machine can model microcode which is Turing powerful —i.e. which may represent partial recursive functions—despite the requirement that an operation define a total function. If a control state is set to Neutral (through the neutralize transition), no other instruction, input, or output which references that state will ever be enabled again, which is exactly the behavior that we wish to attribute to operands which are never granted or reserved due to a divergent computation in micro-code.

Obviously, the micro-code cannot test to see if it will never halt in order to initiate a neutralize transition. Suppose that after the microcode for an instruction has executed for a while and failed to perform grant or reserve transitions for some of its operands, the machine assumes that the microcode has started looping and assumes neutralize transitions for those operands. Suppose, further, that even after performing this action, the machine leaves the microcode running as it initiates the next instruction. Some time later, perhaps after several more execution steps have occurred, the microcode may perform another transition. The machine was wrong to assume that a transition for the operand would not appear.

However, there is no harm done. No execution steps or final markings have occurred that could not have occurred if the machine had waited longer for the transition to appear. So, the machine can perform the late transition and continue normally.

Carrying this principle further, when the machine finds an enabled instruction, it can simply assume neutralize transitions for all of the operands, then begin execution of the microcode and immediately start looking for the next enabled instruction. In fact, this is exactly how the current version of LGDF2 is implemented. This suggests another opportunity for parallel execution: as soon as a transition is performed for an operand, that state is available to play a role in enabling another instructions.

The neutralize transition also comes in handy to model run-time fatal errors in micro-code—as long as the presence of the error is deterministic.

6. Execution Graphs

Although the definition of execution as a sequence of execution steps is suitable for sequential models of computation, it is not suitable for parallel models. Such sequences contain both too little information (e.g. about what dependency relationships existed among the steps) and too much (e.g. the order in which totally unrelated events occurred). Executions that we would probably like to consider equivalent may appear wildly different by looking at the sequences alone.

A more desirable view of execution would be one which described the order of two execution steps only when it might affect the output. Any sequence of execution steps which satisfies such a partial ordering could then be considered as identical, in the sense that all would map inputs through the same instructions and therefore produce the same output [10]. Our attempt to define such a partial ordering will be called an *Execution Graph*, or EG.

The following construction of an EG relies on the fact that an instruction can only directly depend on another if they are connected to the same state in the F-net, since there is no other way for one instruction to affect or detect the relative execution order of another.

The nodes in an execution graph resemble those of the F-net whose execution it is describing. *Value* nodes are shown as squares, and represent the control and data states associated with a given state at a given "time". They are labeled with the address of the state they represent. *Evaluation* nodes are shown as circles, and represent the execution of an instruction on the machine. They are labeled with the opcode of the instruction. Edges will represent control dependence and possibly data flow. When the latter is present, it is indicated by a directed edge. The graph will be built from left to right. The rightmost value node with a given label will be called the *current value node* for that state, since it will represent the most current values on the associated state.

At the beginning of the machine's execution, the execution graph is initialized with a value node for each data memory location in the F-net. When an input is

performed, an arrow is drawn down to the top of the current value node corresponding to the input memory. When an output is performed, an arrow is drawn down from the bottom of the output memory. When an instruction is performed:

(1) An evaluation node for the instruction is added to the right of all of the other nodes in the EG.

(2) Edges are added to connect the new evaluation node to all of the current value nodes corresponding to the operands of the instruction. If an edge corresponds to an operand with **rd** in the permission signature, the edge is directed toward the evaluation node in the EG. Else, it is left undirected.

(3) New current value nodes are added to the EG, to the right of the new evaluation node, corresponding to the operands of the instruction which have either **wr** or **gr** in the permission signature.

(4) Edges are added to connect the evaluation node to the new current value nodes. If an edge corresponds to an operand with **wr** in the permission signature, the edge is directed toward the value node in the EG. Else, it is left undirected.

(5) All edges are numbered for the operand that they represent.

The first element of the output sequence produced by the sample F-net in Figures 1a and 1b will be equal to the sum of the first n cubes, where n is the first element of the input sequence. An execution graph of the sample F-net is shown in Figure 2 for an initial configuration having an input sequence of $\{2\}$.

The execution graph shows the actual transformations that occur to data state during the execution of an F-net. It represents the function which is being simultaneously created and evaluated by the F-net.

The execution graph of the sample F-net is only one of six that could have been obtained from that F-net with that initial configuration. It could therefore be deduced

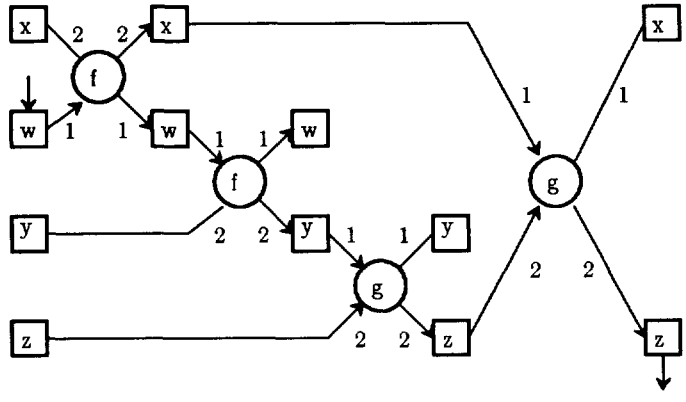

Figure 2. Execution Graph of Sample F-net with input $\{2\}$

that the F-net is non-deterministic, since it can result in different computations for the same input.

It is not too hard, however, to show that all six functions are indeed the same by performing transformations and reductions on the execution graph which preserve the function being represented. The easiest of these transformations is to remove the labels on the value nodes not representing input or output memory. This immediately trims our six graphs to two. To show these two EGs represent equivalent functions requires only that we recognize that the "g" operation is commutative. Other reductions can be performed on EGs, and different definitions of "deterministic" can be formulated based on the equivalence of such "collapsed" execution graphs.

7. Motivation

In this section, I describe the motivation behind various aspects of the design of the VM as though all aspects of its design are new. In fact, many of the points defended here have been present in the LGDF model since its inception, though some may not have been explicitly justified in the past. This section, then, is written with the benefit of 20/20 hindsight!

7.1. Data State

A data state in the VM is a set of variables that are treated as a unit in terms of process access. It can be derived using the idea of shared values—variables present in different processes, perhaps having different names, but which are intended to represent exactly the same concept in each. A sharable value is identified by this concept rather than the variable name it is represented by. The data states of a program can be constructed by partitioning the set of all sharable values by the set of sharable values read by each process, the set of sharable values written by each process, and the set of sharable values updated (written and read) by each process.

Shared memory programming has always relied on the concept of data state to give reasonable meaning to locks; i.e. what is a lock locking? Message-passing programming, too, must always be concerned with which processes have access to which data state, and it's natural to think of identical data used by many such processes as being somehow "the same data". The VM has formalized this notion.

7.2. Control State

Control state unifies the ways that locks and messages are used to control access to data state. Both of these mechanisms can instill an ordering on access to data state, but locks naturally implement partial orderings, where the next process to get access to the lock and therefore the data state is not known, and point-to-point messages naturally instill a linear ordering, where the next process to get access to the message and therefore the data state, is known completely. If process synchronization was limited to either of these extremes in the VM, the program on the VM would need to implement other orderings with an explicit protocol. Control state allows both extremes or anything in between by partitioning the processes that can access the

state into two (or, as will be shown later, more) sets. When a side becomes active, only those processes connected to that side (i.e. within that set) can access the state. The implementation can then provide the most efficient protocol for the ordering required on the available architecture.

7.3. Data-driven Scheduling

An instruction does not explicitly poll for each of its data states to become available. Rather, it implicitly waits for them to all become available. This assurance that all states are available for access when it begins eliminates the possibility of deadlock and makes it possible for an instruction to treat the associated data states like a function would treat its arguments.

7.4. Read and Write Permissions

Efficient sharing of data state on a particular architecture can only be performed when the read or write intentions of each process is known. It is common, in shared memory machines, to consider some area of memory as "read only" during some portion of the program execution. Such an area can be read simultaneously by several processes, but not written by any of them. Since this is not possible if processes are accessing separate memory in a private memory system, the less efficient method of sending a message containing a copy of the data to each process must be used if simultaneous reading is desired.

The message passing solution, though less efficient in most cases, has an advantage; a process can update its copy in some cases even while other processes are still reading old copies of the data. This same effect can be accomplished by a shared memory approach, but only by physically copying the data to a new location for each of the reading processes.

Since the intentions of a process to read or write the data state must be explicit to efficiently implement its sharing and communication on a particular architecture, the combination of read *and* write is a natural extension. This allows a process to update in place, further reducing unnecessary copying (within the user's instruction microcode).

7.5. Dis-Allowance of Neutralize for some Permission Signatures

In order to be architecture independent, microcode must specify the same set of transitions regardless of the implementation. This includes neutralize transitions. This only creates a problem in one case: when a state is readable but not writable by an instruction, and the instruction only performs one possible transition to the state (i.e. grant or reserve). See process B in Figure 3.

An efficient message-passing implementation of the VM could send a message containing the data state to the instruction, perform the allowed transition (in this case, reserve) immediately, and ignore the explicit request from the microcode to perform the transition. This would allow other instructions (A, C, or D) to be enabled at the earliest possible time, maximizing possible parallelism. In fact, if this strategy is used

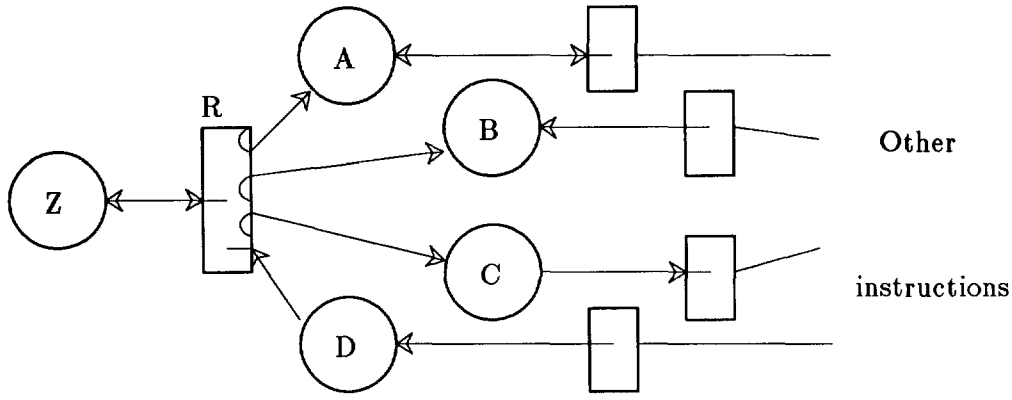

Figure 3. Premature Transition Anomaly

and there are many such enabled instructions on the same side of a state all having read and reserve permissions only and not directly dependent on each other by any other state (e.g. A and C), the contents of the data state can be broadcast to them simultaneously. This natural representation of broadcast within the model is a Very Good Thing.

But if such an instruction should go into a loop before explicitly performing the transition, it is of no consequence; the implementation has already performed the transition. A subsequent instruction which needs to write to the state (e.g. D) could be enabled. Contrast this with a shared memory implementation. The reader's transition cannot be performed until it is explicitly performed by microcode, since that is how a shared-memory implementation keeps writers from intruding on the shared data state while it is being read. If the very same microcode goes into a loop here before performing the transition, it would block writers out forever.

Thus, an efficient shared-memory implementation of a reading process does not have the same semantics as an efficient message-passing implementation. The semantics for the model must be defined one way or the other. Which architecture should lose, and be forced to implement the VM in an inefficient manner?

We declare message-passing the winner, but its not that bad for shared memory, since (1) it is only necessary that a reader perform the transition on the state in a finite time, and (2) a shared-memory implementation can be defined to include a human. Thus, shared-memory reading can be implemented in its writer-blocking manner as long as a human or "gremlin" can check periodically to determine whether a reader is indeed blocking another instruction. If so, the implementation must allow the human or gremlin to manually make a new copy of the data state being read so that the blocked instruction can proceed. An implementation without this feature will not strictly conform to the model, but will still be perfectly usable.

Note that a programmer may know that an instruction is likely to read a state for a long time in a shared-memory implementation, and would prefer that it get a copy of the data state in the first place to avoid blocking writers. This is one of the

implementation decisions that a shared-memory implementation should accept advice on.

8. Extensions (or Higher Level Abstractions)

Is the model powerful enough? One way to answer this is to consider whether extensions to the model add any power or if they can be constructed within the existing model. Even if they can be constructed, it may still be desirable to use them instead their equivalent basic constructs if they are clearer or easier to work with. In this sense, these extensions are similar to control constructs in high-level languages.

Three such extensions are considered in this section.

8.1. Multi-sided States

Consider allowing the model to provide any number of sides on a state, rather than just two. The idea of a transition would need to be extended past grant and reserve to allow the ability to make any side the next active side, but would still include the notion of a neutralize transition and neutral state.

An F-net with an n-sided state can always be modeled by a *simple* F-net (one containing only 2-sided states) with the same number of instructions. To see this, consider a simple F-net with *lg(n)* 2-sided states. There are n ways of attaching processes to one side of each of these 2-sided states. Without loss of generality, consider one of those n ways. If the instructions connected in this way are enabled, no instruction connected in another way can be enabled, since the other instruction must be attached to the other side of at least one of the states, and only one side can be active at a time. By selectively granting and reserving these states, an executing instruction can arbitrarily determine the next active side for all of the states, thereby dictating which connectivity will be enabled next. This is exactly the behavior required for an n-sided state. See figure 4 for an example of a mapping from a 4-sided state to 2 2-sided states. Note that only one of the 2-sided states needs to carry the data state from the

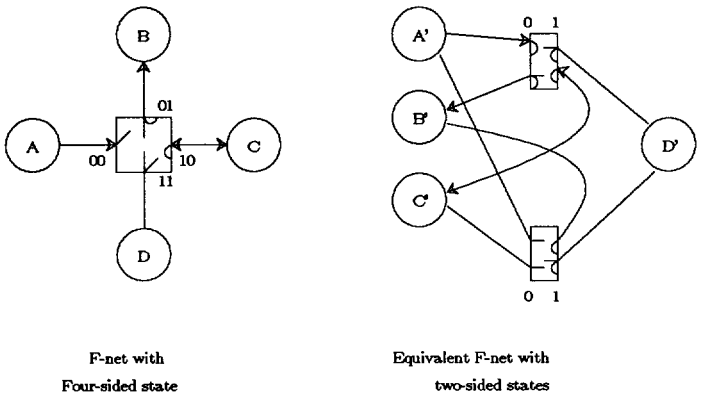

Figure 4. Multisided state reimplemented with 2-sided states

n-sided state.

The introduction of multi-sided states adds to the decision power of the model. Without them, it is often necessary to connect multiple instructions to a single side even when there is no intention for them to be enabled at the same time, simply because they need to access the same data state. Determining the conditions under which these instructions can be enabled is sometimes not clear from the F-net. For example, in figure 4, it is not clear from the simple F-Net which transitions instruction D' may perform, in contrast to D in the other F-Net.

Note that even if an F-net contains multi-sided states, and those states can often only be modeled by 2-sided states by attaching multiple instructions to a side, the determinism rule mentioned earlier still stands: a (multi-sided) F-net which has at most one instruction connected to each side of each state must be deterministic.

8.2. Ignoring Some Control State

In the current model, an operand lists exactly one side of the referenced state which must be active for the instruction to be enabled. Is it possible to have an instruction which can be enabled regardless of the active side of one of its states?

Because an instruction's behavior is defined solely by its opcode and the data states of its operands, this is easily accomplished by providing two instructions which are identical except for the side of the state that they reference. Since only one of the instructions will be enabled at any one time, the behavior of the two combined will be identical to the one instruction which ignored the control state of the state.

8.3. Multiple Active Sides

Even a many-sided state can only have one side active at a time. Would relaxing this rule to allow 2, 3, or n sides active at once, increase the power of the model?

No, since an s-sided state which allows any combination of multiple active sides can be reimplemented as follows:

(1) Replace the original s-sided state with a state having additional sides representing all of the combinations of the original s sides.

(2) For every side j on the new state which represents a combination including side i of the original state, create new instructions that are exactly like those attached to side i except that they reference side j.

See figure 5 for an example of a 3-sided state allowing multiple active sides modelled by a 7-sided state allowing only single active sides.

Although this could be considered a high-level abstraction, it seems that it could add complexity more often that it would remove it.

9. Discussion

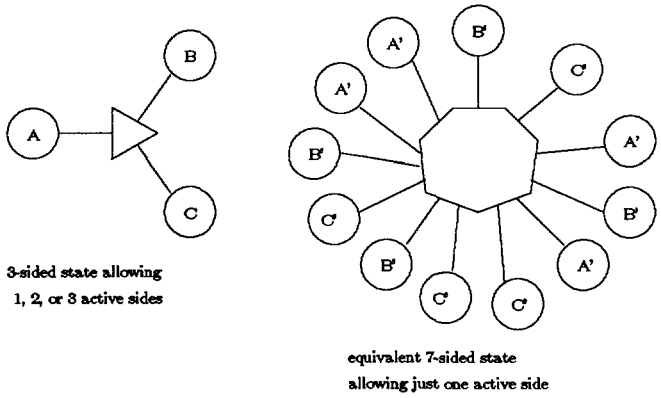

Figure 5. Implementation of multi-active state with uni-active

9.1. Relationship to Petri Nets

Except for the presence of input and output memory, the semantics of a simple F-net resembles that of an *ordinary Petri Net* [11]. In that model, a net consists of directed graph with two types of nodes: places and transitions. Places can be connected only to transitions, transitions only to places. The semantics consist of placing a single token on some subset of the places to form the initial marking, then nondeterministically choosing from among the enabled transitions: that is, ones for which all the input places (connected to the transition by an edge toward the transition) have a token and all the output places (connected to the transition by an edge away from the transition) have no token. When such a transition is chosen, it is fired; i.e., the tokens are removed from all of its input places and a token is placed onto all of its output places. Another enabled transition is then chosen, ad infinitum or until there are no more enabled transitions.

We replace the places in Petri's model with states, transitions with instructions, and tokens with control state (say, Right==token, Left==none). We change the notation to reflect the flow of data rather than the flow of tokens: a state connected to an instruction via its right side would be considered an "input place" in Petri terms, one connected via its left would be an "output place". The semantics of our model differs from Petri's in our use of data. We add data to our states where Petri had none, and extend the "firing" of an instruction by having it decide whether to "pick up or put down tokens" based and the data present on the states. Finally, we produce new data for the states as a function of the data that was on them when the instruction started.

An F-net which does not contain `rd` or `wr` in its permission signatures can be modeled directly by such a Petri Net. See figure 6 for an example. As long as all permission signatures contain only `gr`, the mapping between the two models is purely notational. If `rs` permissions appear in the F-net, the Petri Net must model the associated control state with two places (left and right) and be augmented with an additional transition to copy the token back to the appropriate place when it is reserved.

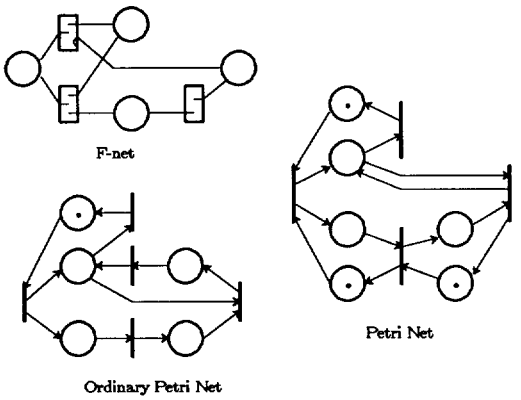

Figure 6. F-Net modeled by Petri Net

Modeling an F-net with a *full* Petri Net (i.e., one which does not require that output places be empty for a transition to be enabled) always requires two places per state, but F-nets with `rs` in their permission signatures can be modeled without extra transitions.

We can benefit from some of the extensive study on Petri Nets, specifically that on reachability. This is the property that some given object marking (in our case, global control state) can or cannot be reached from a source marking. This property has been shown to be decidable for Petri Nets, and algorithms have been developed and proven. Its most obvious use is to determine whether some undesirable global control state can be reached from, say, an initial marking. Unfortunately, our more powerful concept of "firing" limits the usefulness of the algorithms, but they can probably still be used directly on those portions of the F-net where the instructions have only one allowable transition.

9.2. Relationship to Side Effects and Functional Programming

In traditional dataflow[12] or functional programming models, a function always maps a set of inputs to different output(s). There is no distinction within these models between output that will be identical to some input unless altered and that which will not. This additional information is present in F-nets through the use of arcs which are both readable and writable. It can be used to create more efficient implementations by allowing the functions to alter the data in place, where this is possible and desirable.

The problem is that a state may now "hold" different kinds of values, having different characteristics, at different stages of the overall computation. This is the same problem which has greatly complicated attempts to reason about sequential programs containing side-effects, and which has given the impetus to functional programming as an "easier to reason about" paradigm. Rather than give up this idea of in-place data modification, however, we have provided each data state with a control state which represents the characteristics of the data item now inhabiting the state.

The semantics of the machine guarantee that a data item only be accessible to a given function when the "right" type of data item is inhabiting the state. The use of multi-sided states allow full expressiveness of this interpretation of control state.

Data items within a given state can be considered as "useless" (uninitialized or has otherwise outlived its usefulness), "new" (put into the state by a function which does not have read access to the state) or as "modified" (the result of a function modifying a new or modified state). The control state can describe the stage of data state. Thus, the overall state of a computation can be considered as combination of the data items present and the stage of completion of each. A function serves to construct new values from existing ones, to transform existing values in place, or a combination of the two.

This is different from the way a sequential programmer would think of the current state of a computation because the only real control state available is the program counter, and the only way to infer the stage of completion of any variable is to reason about the current program counter and some combination of other variable values. In that sense, the F-net model can be considered as distributing the program counter, not over the program counters of the individual processes, but rather over the "stages of completion" of the variables.

10. Summary

This paper has dealt with the computational aspects of the LGDF2 model in an attempt to provide a formal framework to reason about programs. It has avoided specific recommendations for optimization of implementation in favor of building a model with enough expressibility to make these optimizations possible. It has also not dealt with language issues such as syntax, typing, or modularity. These must all be dealt with in their own right to make LGDF2 a truly useful tool. All of these matters should be considered in terms of the language, however, and few should affect the model.

The model still requires some enhancement to encompass some of the requirements of portable parallel programming. Among these are the ability to partition "arrays" among individual instructions, and the ability to alter an F-net during execution. Both of these are currently being carefully considered. It appears that the next extension will be the addition of an optional indexing field to the operands of any instruction. This will require some data memory to have additional constraints placed on its addresses.

Methods of collapsing and analyzing execution graphs need to be formalized. Besides helping in the proof of correctness of parallel programs, execution graphs are useful in the debugging of parallel programs, since they summarize all the important information about a parallel execution without overconstraining that notion. We are developing a graphical debugger [13] which rebuilds an execution graph by recording minimal data during an actual execution, then re-executes the program in an "Instant Replay" [14] fashion.

The two-level approach to parallel programming presented here seems to be valuable. This is not a new concept, but the F-net model can be considered as an attempt to see how few restrictions can be placed on behavior of the component processes and the overall program while preserving the abstraction of the processes.

11. References

[1] T. DeMarco, "Structured Analysis and System Specification". New York: Yourdon, 1978.

[2] R. G. Babb II, "Programming the HEP with Large-Grain Data Flow Techniques", in *MIMD Computation: HEP Supercomputer and Its Applications,* (ed. by J. S. Kowalik). Cambridge, MA: The MIT Press, 1985.

[3] D. DiNucci, R. Babb II, "Practical Support for Parallel Programming", in *Proc. 21st Hawaii Int. Conf. on System Sciences,* vol. II, pp. 109-118, (ed. by Bruce Shriver), Computer Society Press of IEEE, 1988.

[4] N. Carreiro and D. Gelernter, "The S/Net's Linda Kernel", *ACM Trans. on Computer Systems,* vol. 4, no. 2, May 1986, pp. 110-129.

[5] C. A. R. Hoare, "Monitors: an operating system structuring concept", *Communications of the ACM,* vol. 17, no. 10, pp. 549-557, October 1974.

[6] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, R. Stevens, "Portable Programs for Parallel Processors". New York: Holt, Rinehart and Winston, 1987.

[7] J. R. McGraw, et al., SISAL: Streams and Iterations in a Single-Assignment Language, Language Reference Manual, Version 1.1", U. of California, Livermore Livermore National Laboratory, Technical Report M-146, July 1983.

[8] J. B. Dennis, First Version of a Data Flow Procedure Language," in *Lecture Notes in Computer Science,* Vol. 19, Springer-Verlag, 1974, pp. 362-376.

[9] Gilles Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP74,* North-Holland, 1974, pp. 471—475.

[10] V. Pratt, "Modeling Concurrency with Partial Orders", *Int'l Journal of Parallel Programming,* vol. 15, no. 1, Feb. 1986, pp. 33-71.

[11] J. L. Peterson, "Petri Net Theory and the Modeling of System", Englewood Cliffs, NJ: Prentice-Hall, 1981.

[12] J. A. Sharp, "Data flow computing". Chichester, West Sussex, England: Ellis Horwood Limited, 1985.

[13] D. DiNucci, "High-Level Parallel Debugging with LGDF2", *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* 1988.

[14] T. LeBlanc, J. Mellor-Crummey, "Debugging Parallel Programs With Instant Replay", *IEEE Transactions on Computers,* vol. C-36, no. 4, Apr 1987, pp. 471-482.