Tolerant (Parallel) Programming

David C. DiNucci
MRJ Technology Solutions, Inc.
NASA Ames Research Center, M/S T27A-2
Moffett Field, CA 94035

Abstract - In order to be truly portable, a program must be tolerant of a wide range of development and execution environments, and a parallel program is just one which must be tolerant of a very wide range. This paper first defines the term "tolerant programming", then describes many layers of tools to accomplish it. The primary focus is on F-Nets, a formal model for expressing computation as a folded partial-ordering of operations, thereby providing an architecture-independent expression of tolerant parallel algorithms. For implementing F-Nets, Cooperative Data Sharing (CDS) is a subroutine package for implementing communication efficiently in a large number of environments (e.g. shared memory and message passing). Software Cabling (SC), a very-high-level graphical programming language for building large F-Nets, possesses many of the features normally expected from today's computer languages (e.g. data abstraction, array operations). Finally, $L2^3$ is a CASE tool which facilitates the construction, compilation, execution, and debugging of SC programs.

I. Introduction

The subgoals of parallel processing are very similar to the subgoals of software engineering in general—i.e. the decomposition of a large problem into smaller tasks or modules, the precise expression of the scope of data and the semantics of sharing and communicating data efficiently and safely between modules. In fact, differentiating parallel software engineering from traditional software engineering can be a mistake, since doing so may lead one to believe that a parallel program is engineered in a different (and perhaps even more roundabout) way than a "real" program. Parallel software engineering will only come into its own when parallel programs are considered "real", and the ability for a program to run efficiently on a parallel machine is just another desirable feature which the software possesses. A corollary is that tools, languages, and methodologies for "parallel" programming must be useful enough to facilitate any kind of programming.

This may seem like a lot to ask. After all, parallel software engineering seems to lag behind sequential in virtually every area: formal models, languages, tools, development and debugging strategies. Of these, formal models constitute the linchpin. With an adequate formal model, the other methodologies can be developed to exploit the properties of that model. To facilitate the needs above, the model must be tolerant of, but not dependent upon, the sharing of memory or the passing of messages. Likewise, it must be adaptable to different languages, different architectures, different amounts of parallelism (including none at all), and the possible presence of nondeterminism.

The objective of this paper is to present a tolerant approach to software engineering. That is, rather than engineering a product for a particular parallel environment, the goal is to engineer one which is tolerant of many different environments, including parallel environments. The paper will first describe the full meaning of tolerant programming, then will present a simple yet formal model for (parallel) computation, called F-Nets, which embodies that meaning and thereby serves as the basis for a host of other tools. Finally, a brief description of some of those tools is given, followed by some conclusions.

II. Tolerant Programming

Although the term "tolerant programming" could be used to describe tolerance to any number of traits, those favored here are concurrency, latency, semantics, changing environment, bugs, and language. Tolerance of these traits refers to the ability of a program or programming methodology to work well independently of the values of these traits. Some techniques which might be used to obtain tolerance are described under each heading, but tolerance may be achieved through either user effort or automatically through tools (e.g. compilers). In fact, tolerance often relies on the ability to infuse the program with potentially useful information, and this may not be possible without the aid of tools.

A. Concurrency

Concurrency tolerance refers to providing the opportunity for scaling while not sacrificing efficiency on less-scalable platforms. For example, a large program can be decomposed into relatively small parts which may be able to execute concurrently, but there should also be a mechanism to efficiently recompose these into larger portions when parallel execution is not possible.

To express program decomposition, constructs which are well defined and easily understood in both sequential and parallel environments should be used. Traditionally, loops have filled this role, but there are better examples, such as functions and atomic transactions.

B. Latency

Latency tolerance refers to the ability to adapt to high-latency environments without impacting efficiency in low-latency environments. Such tolerance is gained primarily by providing information about the expected data movement patterns. For example, whenever possible, the programmer should:

1) Express need for data long before it is used (and maybe even before process starts running—i.e. staging),

2) Express where data will be used next, even before it is requested there,

3) Refrain from waiting for last version of data to be consumed before creating next version (i.e. queueing),

4) Block (group) data to allow multiple data items to move as a unit whenever it is requested or forwarded, thus cutting latency/byte,

5) Pipeline computation.

C. Semantics

Semantic tolerance refers to the recognition that different environments may have different default semantics, so the desired semantics (or lack thereof) should be clearly expressed. For example, shared memory and message passing are different standard semantic combinations for communication, and using one combination globally throughout the program can make the program much less efficient in an environment which implements the other by default. A better approach is to delineate the application's semantics for each data communication, so that the required semantics can be implemented in the most efficient way in the available environment—e.g.

1) Destructive read (i.e. read on non-empty, dequeue)

2) Destructive write (i.e. standard over-write)

3) Non-destructive write (i.e. enqueue), or

4) Non-destructive read (i.e. standard read)

In some cases, it is appropriate to express the lack of a specific required semantics, especially as it relates to the ordering of operations. For example, a designer may not care about the order in which some commutative arithmetic operations are performed, nor perhaps the order in which some outputs are produced. Such an expression of acceptable non-determinism may make a program run faster in environments where loads or processor speeds vary. It is important, however, to ensure that non-determinism does not sneak into the design.

D. Changing Environment

Changing Environment tolerance refers to adaptation to an environment which changes as the program is running. Environment Tolerance includes the well-known Fault Tolerance, which is tolerance to processors which are made unavailable in an abrupt and unannounced fashion, but it also includes tolerance to any circumstances when processors leave in a preannounced and well-defined way, and/or when processors become available which were not previously present. This tolerance takes that the computer(s) will likely be shared by many users, so the programming and/or scheduling method should allow for efficient sharing of limited resources.

E. Bugs

Bug tolerance relates to the provision of adequate debugging tools and mechanisms which provide the ability to find bugs and which limit the scope of bugs if they exist so that a small bug in one part of the program will have limited effects in other parts. This includes the ability to find unwanted non-determinism and to efficiently record desired non-deterministic choices made during execution of a parallel program so that it can be reliably debugged and analyzed in a cyclic manner.

F. Language

Language tolerance describes the ability of a programming methodology to retain its utility even when the specific programming language used for implementation changes. This allows the implementor the flexibility to use the appropriate language for the job, or to change the language for any number of reasons, without requiring totally different engineering methodologies.

III. F-Nets

This section will informally describe a model for parallel computation called F-Nets. Although this model is not widely known, it is similar in many ways to Actors[1], Petri Nets[10], Turing Machines, Finite State Machines, dataflow, and CCS[9], all of which are well known. It was originally developed as a refinement of the Large-Grain Data Flow (LGDF) parallel programming approach[2], and early versions went by the name LGDF2 [6]. A full, formal description (containing an axiomatic semantics) can be found in the author's Ph.D. dissertation[5], although the version here has been somewhat simplified (i.e. by merging the "instruction" and "operation" abstractions). This paper will present only an operational semantics, and using only informal English, but this will be sufficient to illustrate the value of the model.

The model will be described in four parts: its syntax, its semantics, its efficient expression on real computers, and some favorable practical and theoretical properties it possesses which relate to tolerant programming.

A. Syntax

An F-Net can be considered as being similar to a Turing Machine, having a (possibly-infinite) tape, separated into *squares*. Each square contains a mutable value, called the *data state* of the square, and a mutable color, called its *control* *state*. Call the set of all possible data states S.

Along with the tape, the F-Net consists of a possibly-infinite set of *transitions*. Associated with each transition is a set of colored (non-white) *heads*. Each head is permanently attached to one square—i.e. the tape does not move relative to the heads. Each head is either a `read` head, a `write` head, a `read-write` head, or a `nodata` head (i.e. a head having neither read nor write capability). The heads for a given transition are enumerated from 1 to n, where n can be different for each transition. No two heads from the same transition are attached to the same square.

Each transition has an associated *firing function*, which can be visualized as a table. If the number of `read` and `read-write` heads attached to the transition is r, the table has $r^{|S|}$ entries—i.e. one entry for each possible combination of r symbols under those heads. Each entry contains n colored (possibly white) symbols—i.e. one symbol corresponding to each head.

Any of the `read` and `nodata` heads can be declared "predictable c" which is a declaration that the symbol corresponding to the head in all of the firing function entries has the color c.

B. Operational Semantics

An F-Net works as follows. The machine begins in an initial state consisting of a predetermined symbol and the color green to each square. Then, repeatedly, a *ready* transition is located (subject to the *liveness/fairness* rule below) and *evaluated* until there are no more ready transitions. A ready transition is defined as one for which each head is the same color as the square to which it is attached. Evaluation (or *firing*) consists of finding the entry in the table corresponding to the symbols under the `read` and `read-write` heads, and replacing the color of the squares under all heads with the color of the corresponding symbol from the table entry. For each write and read-write head, the symbol (i.e. data state) of the square is also changed to the corresponding symbol from the table entry.

The liveness/fairness rule states that if a transition is ready, then either it or some other ready transition which shares a square with it must be evaluated eventually (i.e. will not be eternally preempted) in the repeat cycle described above.

Note that for `read` and `nodata` heads, the symbols in the table serve no purpose other than as place-holders for the color at those positions, and that for predictable heads, even the colors in the table are superfluous. Note also that if any square becomes colored white during the course of an F-Net execution, no transitions attached to that square will ever be ready again, since heads cannot be white.

There is never a need to consider the relative ordering of tape squares, so F-Nets are often represented graphically with the squares separated into disjoint rectangles. Each transition is shown by a circle, and each head by a colored line between the transition circle and the tape square rectangle. Arrowheads denote whether the head is `read-write` (both ends), `read` (transition end), `write` (square end), or `nodata` (no arrowheads). Although the firing function is rarely represented graphically, the colors which might be assigned to each head are often represented by colored dots near the connection between the line (head) and the tape square (rectangle). `Read` and `nodata` heads with only one colored dot are implied to be predictable.

C. Real-World Implementation

To understand how the F-Nets model pertains to tolerant programming, it is first important to understand how its components manifest themselves in the real world. Only two such components will be discussed here: the firing functions and the tape squares. The remainder of the F-Net is typically represented in the same graphical form previously described.

In a standard computer, the firing function of each transition of an F-Net is normally implemented as a small deterministic (usually sequential) subprogram. That is, instead of being a table, the function is represented by the mapping of inputs to outputs implied by the code. The data state of each square is implemented as a standard data structure. This leaves only the colors (i.e. control state) to be handled in a somewhat non-standard manner.

When a transition fires, the data state of the squares under the transition's heads are passed to the subprogram as arguments. Although `nodata` heads can be neither read nor written, they are also supplied as a special kind of argument. At any time during the execution, the subprogram can execute a special statement, of roughly the form

```
give square color
```

which declares that the subprogram will make no further accesses to argument *`square`*, and that the square should be assigned the new color *`color`*. By executing one such statement for each of its arguments (i.e. squares), the subprogram expresses a mapping from the initial state of its read and read-write squares to the final state of its `read` and `read-write` squares and the new color for all of its squares, as it is required to do by the formal F-Net semantics. If a transition subprogram does not execute a `give` statement for some of its arguments, those squares effectively become white.

The word "repeatedly" in the semantics (third sentence) suggests that only one transition can be evaluating at a time, but this leads to both practical and theoretical problems. Practically, requiring sequential execution obviously decreases the model's value in the realm of parallel processing. Theoretically, sequential execution would require that a scheduler know when the evaluation of a transition subprogram was finished so that it could know that the next ready transition could be initiated. This means that, at every point in time, the scheduler would be required to decide whether further evaluation of the subprogram might lead to execution of more `give` statements (i.e. for arguments for which they had not already been executed). In other words, a correct sequential scheduler would be required to either solve the (impossible to solve) halting problem, or to conceivably let subprograms which never execute `give` statements for some arguments execute forever and thus contradict the required liveness properties.

Fortunately, the stated semantics can be shown to be identical to these revised semantics:

> "An F-Net works as follows. The machine begins in an initial state .... Then, repeatedly, a ready transition is located and *initiated.* A ready transition is ..... Initiation means changing the color of all the squares under the transition's heads to white, and then beginning evaluation of the transition. Evaluation ..."

In this case, the scheduler does not need to wait for one transition to finish its evaluation before initiating the next. It yields the same result as the previously-stated semantics because each transition is atomic by virtue of being a two-phase transaction[8]: it has a growing phase where it acquires all of its resources (i.e. changes all of its squares to white, effectively locking them), followed by a shrinking phase where it relinquishes them (i.e. gives them a color). Although two-phase transactions can sometimes deadlock, the fact that the revised semantics still initiates the transitions "repeatedly" avoids this problem, and even this sequentiality can be avoided as long as the initiation of a transition is ensured to be an atomic action (e.g. by enclosing it in a critical section covered by a lock), or by changing the color of the tape squares to white in a predetermined global order.

Predictable heads can provide a great opportunity for optimization. Since only `read` and `nodata` heads are predictable (by definition), the new data state and control state for the associated square is known the instant the transition fires. This means that the scheduler itself can pre-`give` the square a color during the scheduling process and immediately schedule other transitions which become ready as a result.

The control state (color) of each square is implemented as some internal state which allows a scheduler to determine which transitions (subprograms) to schedule. It is often most efficient to distribute this control state among the transitions. That is, rather than representing the control state of each square in a particular location, each transition is given a "reasons count"—i.e. an integer which describes the number of the transition's heads which are over the wrong color of square. Each time a transition is initiated or executes a `give` statement, the appropriate reasons counts are adjusted (within a critical section), and new transitions are scheduled (i.e. initiated) whenever their reasons counts reach zero[7].

In distributed memory environments, `give` statements associated with `write` or `read-write` heads often map straightforwardly into message `sends` which pass the data state associated with the square to the next process that will read it. There are, however, some circumstances where the next process to read the data cannot be immediately. In that case, the data state can either be left with the transition to be requested later when the reader is finally determined by the scheduler, or it can be forwarded to a location which is physically closer to all potential readers.

D. Tolerance and Other Properties of F-Nets

The visual nature of F-Nets springs from the nature of computation and the relationship between algorithms and computations. In the sequential world, a computation is usually considered as a sequence of operations. One possible algorithm to express a particular sequential computation is a straight-line algorithm which simply performs each of the operations in the proper order, but the power of programming is obtained by "folding up" this straight-line algorithm with loops and conditionals, so that it takes much less space. During execution, such a "folded up" algorithm both performs the operations designated therein and unfolds into a sequence at the same time, and the unfolding itself can be affected by the inputs provided to the algorithm.

Similarly, a parallel computation is often considered as a partial ordering of operations [11]. However, the term "parallel algorithm" has heretofore not had a very formal definition. An F-Net algorithm is an almost perfect analog to a sequential algorithm—i.e. it is a folded up partial ordering of operations, which is unfolded as it is executed. This "folded partial ordering" description explains why F-Nets are represented most naturally as graphs.

Unlike sequential algorithms, some F-Net algorithms may unfold into different partial orderings, even when given the same inputs (or in this case, initial markings). This nondeterminism is a desirable characteristic, as described earlier under semantic tolerance, as long as it is not introduced by accident. Potential non-determinism in an F-Net can be detected syntactically, so tools can allow the user to verify that it is desired. Specifically, an F-Net may be non-deterministic if and only if it contains two (or more) transitions which have like-color heads on the same square (say *s1*) and those same transitions do not have differing-color heads on another "shared" square (say *s2*). The non-deterministic choices made during execution can be recorded efficiently by just recording the order in which the stated transitions fire—i.e. one bit recorded for each execution of the offending transitions—and this information can be used during debugging to ensure repeatability.

Language tolerance is achieved in F-Nets because the model requires only that each firing function represent a deterministic mapping from some set of data values to some new set of data values and to a color for each head. The representation of this mapping is not restricted: e.g. it can be in the form of an imperative subroutine, as described, or in terms of a functional, dataflow, or logic program fragment[1]. This pro-

1. Note, however, that F-Nets support update-in-place through the use of `read-write` heads, so using other paradigms may inflict additional copying in some cases when using these heads.

vides maximum flexibility to use any language, and even to use different languages for different transitions.

The fact that each transition represents a simple mapping, independent of anything else going on at the time, is indicated by the (first) semantics. That is, even though transitions may execute concurrently, they must act as though they are executing one by one. This not only provides concurrency tolerance, since the constructs being used have identical behavior in parallel and sequential environments, but also bug tolerance, since errors in implementing a transition can only lead to errors in the mapping represented by that transition. Moreover, since the semantics of the language used to implement the transitions does not change in a parallel environment[1], standard sequential debugging tools can be used to debug the transition mappings. This is in marked contrast to traditional shared-memory or message-passing programming, where the behavior of any one program or program fragment can only be described by including the possible asynchronous arrival of messages and/or data, and therefore by including all possible global states of the system.

F-Nets achieve latency tolerance through all of the techniques mentioned in the previous section. Since each transition is endowed with the knowledge of the data that it will need to perform its task, this data can be forwarded (staged) by the scheduler to the processor which will execute the firing function, even before the function executes. Latency is amortized by communicating an entire tape square (which could comprise a large data structure) at a time, leaving fine-grain access to its components to occur in a low-latency environment.

Queuing of multiple versions of a tape square is also supported by the model, due to predictable heads. Say, for example, that one transition has a green "predictable red" `read` head on a tape square, and another has a red `write` head on the same square which changes the color to green. When the first transition fires, the scheduler has the option of immediately changing the square color to red, allowing the second (writing) transition to execute again even while the reader continues to execute. Of course, in this case, the scheduler must ensure that the writer uses a separate memory area to create the "next" version of the data state for the tape square.

Transitions are extremely tolerant to both concurrency and dynamic environment considerations due to their atomic, stateless properties. Specifically, by ensuring that each transition is relatively small, an algorithm can expand into any number of available processors, and problems related to the loss or migration of execution state can be avoided completely by backing out of partially-executed transitions. Nevertheless, unlike some functional and dataflow models where data must be copied from one actor (function, program, chare, etc.) to the next, executing multiple sequentially-composed transitions on the same processor adds virtually no overhead above a standard subroutine-call interface.

1. Actually, inclusion within an F-Net may increase in the number of possible error conditions in a program fragment—e.g. when access is made to an argument after a give statement has already executed for it—and this is technically a semantic difference.

IV. Making F-Nets Practical

Even if the theoretical model of F-Nets has some desirable properties, there remain the practical tasks of implementing F-Nets efficiently on real machines and implementing large programs in F-Nets. These tasks require the availability of languages, tools, and methodologies, some of which are briefly described here. A low-level subroutine library, called Cooperative Data Sharing (CDS), which facilitates the efficient implementation of the F-Net model on real architectures will be briefly described, followed by some features of a high-level graphical programming language, Software Cabling (SC), which facilitates the construction of large F-Nets. Finally, a CASE tool called $L2^3$ will be mentioned which is used to design, implement, compile, execute, and debug SC programs.

A. Cooperative Data Sharing

Cooperative Data Sharing (CDS) is a subroutine package which embodies the communication requirements of F-Nets without the dataflow-like run-time semantics[4]. Although a major goal of CDS was to aid in the implementation of F-Nets on real computers, it can also be used as a standard communication substrate for a variety of other purposes. In message-passing environments, it serves a role much like MPI and PVM, but it also works efficiently in shared-memory environments because of the lack of copying allowed by its semantics[3].

In CDS, F-Net data states are implemented as regions of specially-tagged dynamic memory. Pointers to these regions can be passed between any processes. CDS automatically manages the remapping and queuing of these pointers, the migration of regions if pointers are passed between remote processes, the sharing of regions by multiple readers on the same processor, the fetching of regions from remote processors, and the initiation of handlers based upon pointer manipulation. While CDS has been implemented on top of UDP, other projects such as U-Net [12] suggest that extremely efficient lover-level implementation is possible.

B. Software Cabling

Software Cabling (SC) is a visual programming language for building very large F-Nets. An SC program effectively compiles into an F-Net while ensuring that the correspondence between the SC program and the F-Net is always apparent. This allows SC to inherit many of the desirable properties of the F-Nets model while compensating for some of F-Net's apparent deficiencies for "real" programming.

F-Nets do not have a good notion of input and output. An

SC program dynamically becomes part of the surrounding operating system, thus allowing it to access I/O devices and buffers as special tape squares.

An F-Net is just a flat network of squares and transitions, and lacks any notion of modularity (other than the connotation of each transition as a module). SC provides mechanisms for hierarchically composing an F-Net out of smaller networks and allows the ability to "clone" these sub-networks. In fact, since such a sub-network contains program fragments (transitions) which can be initiated by data (tape squares) outside of the network, and also contains data (tape squares) which are not accessible from the outside, these sub-networks effectively form abstract data types, or classes, leading to an object-oriented coding style, usable even when the transitions are implemented in languages such as Fortran 77.

An F-Net is a static construct. This fact would seem to interfere with its ability to adequately express data parallelism and other parallelism which changes dynamically based upon data. It would also seem to interfere with changing dependence structures which can result from the inter-relationships of array subscripts. SC can express dynamic parallelism (i.e. the ability of F-Net fragments to disappear or be replicated) by recognizing that an F-Net is a conceivably infinite construct, and dynamic portions can be considered as parts of the F-Net that were always statically present but were not able to execute due to the color of some tape squares. SC supports dynamically-sized arrays and array subscripting using the same approach—i.e. by modeling each array element or array section as an individual tape square and ensuring that access to that square is in effect controlled by the color that square (or another).

C. $L2^3$

$L2^3$ is a toolset (under construction) used to design, implement, compile, execute, and debug SC programs[1]. It is centered around a graphical editor, which must necessarily be as utilitarian as the standard text editors used to maintain textual programs. Debugging is to be supported both at the level of the F-Net (i.e. by animating the graphical SC program) and the level of individual processes (i.e. using standard sequential debuggers). Finding non-determinism, and instrumenting and replaying non-deterministic programs, is also being supported. Network fragments can also be tested and debugged individually.

V. Conclusion

Tolerant programming is possible. Given a satisfactory theoretical model as a basis, many of the difficulties related to parallel programming can be surmounted. F-Nets not only provides a formal and natural expression for parallel algorithms, but the resulting tools can benefit all programming by providing language and platform interoperability and by providing formal methods and notations for program design and implementation.

1. The name of this tool, pronounced "ell two three", is a simplification of the name "parallel tools" = "pair ell ell two ells" = (LL LL)(LL) = $(LL)^3$ = $L2^3$

Acknowledgment

The idea of "tolerant programming" benefited from discussions with colleagues Robert Hood and Louis Lopez at NASA Ames. The F-Nets model was developed during my studies at Oregon Graduate Institute, where it began with the research and helpful feedback of my advisor Robert Babb, and was furthered by feedback from my committee (Michael Wolfe, Harry Jordan, and Richard Kieburtz) and early discussions with Dick Hamlet. The CDS package was developed at NASA Ames Research Center, where support was provided by the NAS program and HPCC.

References

[1] G. Agha, "Actors: A model of concurrent computation in distributed systems", MIT Press, 1986.
[2] R. G. Babb II and D. C. DiNucci, "Design and implementation of parallel algorithms with Large-Grain Data Flow", in The Characteristics of Parallel Algorithms, Jamieson and Douglass (ed.), Cambridge, MA, MIT Press, 1987, pp. 335-349.
[3] D. C. DiNucci, "A simple and efficient process and communication abstraction for network operating systems", LNCS vol. 1199 (CANPC'97 Proceedings), pp.31-45, Berlin, Springer-Verlag, 1997, pp. 31-45.
[4] D. C. DiNucci, "CDS", http://www.nas.nasa.gov/NAS/Tools/Projects/CDS
[5] D. C. DiNucci, "A formal model for architecture-independent parallel software engineering", Ph.D. Dissertation, Oregon Graduate Institute, 1991, also available at ftp://ftp.netcom.com/pub/di/dinucci/thesis.ps.Z
[6] D. C. DiNucci and R. G. Babb II, "Design and implementation of parallel programs with LGDF2", COMPCON'89, San Francisco, 1989, pp. 102-107.
[7] D. C. DiNucci and R. G. Babb II, "Practical support for parallel programming", Proc. 21st HICSS Software Track, 1988, IEEE, 109-118.
[8] K. P. Eswaran et al, "The notions of consistency and predicate locks in a database system", CACM, vol. 19, 11 (November 1976), pp. 624-633.
[9] R. Milner, "A calculus of communicating systems", Lecture Notes of Computer Science, vol. 92, Berlin, Springer-Verlag, 1980.
[10] J. Peterson, "Petri net theory and the modeling of systems", Englewood Cliffs, NJ, Prentice-Hall, 1981.
[11] V. R. Pratt, "Modeling concurrency with partial orders", International Journal of Parallel Programming, vol. 15, 1(February 1986), pp. 33-71.
[12] T. vonEicken, "U-Net: A user-level network interface for parallel and distributed computing", in ACM Symp. on Operating System Principles, Copper Mountain, CO, Dec. 1995, pp. 303-316.