IMPLEMENTATION OF COMPILER, VIEWER, AND PARALLELISM ANALYSIS SOFTWARE FOR THE IF1 LANGUAGE

IMPLEMENTATION OF COMPILER, VIEWER, AND PARALLELISM ANALYSIS SOFTWARE FOR THE IF1 LANGUAGE

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

SUWANTO, B. S.
Iowa State University, 1994

May 1997
University of Arkansas

This thesis is approved for
recommendation to the
Graduate Council

Thesis Director:

____
(Dr. Mitchell A. Thornton)

Thesis Committee:

____
(Dr. David L. Andrews)

____
(Dr. Carl D. Bowling)

____
(Dr. Mitchell A. Thornton)

Acknowledgments

I would like to express my sincere thanks to everyone who contributed to this research project directly or indirectly. Foremost on that list Dr. Mitch Thornton, my major advisor, for his invaluable suggestions, patience and encouragement throughout the duration of this research. This paper will not be possible without your help. Special thanks also go to other members of my thesis committee, Dr. David Andrews, Dr. Carl Bowling, and Dr. Ronald Skeith for their help, support and advice. I also thank Dr. Daniel Berleant for proof-reading the draft.

Table of Contents

Abstract

The IF1 language has been chosen as the candidate intermediate form for a newly proposed computer architecture, a Multithreaded Parallel Processor. This paper will discuss the process of building the IF1 compiler with a brief description of the new architecture. Additionally parallelism analysis tools were developed and are described. The graphical viewer tools allow the data dependency graph to be displayed. Other tools traverse the graph and calculate data that estimates the exploitable parallelism.

1. Introduction

There are many programming languages to choose from, each with its own capability, appeal, and look to the programmer. Some languages are particularly useful for specific purposes or computers. One unique intermediate language is IF1[1]. IF1 is based on directed acyclic graphs that depict inherent data dependencies. Unlike other languages, IF1 code does not explicitly imply a sequence of instructions to be executed, rather, it defines a data dependency graph which visually depicts the flow of data in a program.

This paper describes the construction of an IF1 compiler and associated analysis tools. The IF1 graph viewer is discussed in one chapter for both Microsoft Windows[1] system and X-Windows environment. Also a discussion on a multiprocessor for simulating the scheduling of instructions based on a Multithreaded parallel processing Architecture[3][4] will be presented. Finally, the conclusion and further possible enhancements provided in the conclusion section.

1.1 Background

Although it is possible to write a program directly in IF1 code, the language was designed to be a low level intermediate form generated by a high level language compiler. As an example, the Sisal (Streams and Iterations in a Single Assignment Language)[2] compiler developed at Lawrence Livermore National Laboratory utilizes IF1 code. Sisal is

[1] Microsoft Windows is registered trademark of Microsoft Corporation.

a functional language that takes advantage of parallelism in a program by compiling into a data dependency graph represented by IF1. IF1 provides a convenient programming medium for expressing large-scale scientific processes for execution on multiprocessor systems comprising hundreds, even thousands, of processors. The Sisal programming environment currently consists of a compiler, a debugger or interpreter, a profiler, and other tools. It can automatically correct a variety of syntax errors and performs a number of optimizations.

Unlike other languages, with Sisal, a programmer can indicate to the compiler instructions to be executed in parallel. The Sisal compiler then utilizes this information to generate IF1 code which is further translated to machine language.

To see the inherent parallelism in a program, it is convenient to view it visually by plotting the number of concurrent executing instructions versus the current clock cycle. A program that uses an IF1 graph to produce such a plot is included in the work described here.

1.2 Motivation

The IF1 compiler development effort described here is part of an ongoing project to develop a *Multithreaded Parallel Processing Architecture*. This project requires a data structure that represents the data dependency in a program. Thus, there is a need for a compiler that will parse an IF1 code and generate such a data structure. In addition, an IF1 graph viewer is found to be helpful to assist the programmer in debugging a program and verifying results. Once the data structure has been created, graph traversal algorithms are used to estimate parallelism and execution time parameters.

1.2.1 Multithreaded Parallel Processing Architecture

This architecture is designed to exploit available parallelism in a program. Sections of code that are sequential may be scheduled as a single thread and sections of the code that are rich in parallelism may be partitioned into many concurrent threads. Figure 1.2.1.1 shows the diagram of the architecture.

The architecture uses two multiprocessor units, the graph engine (GE) and the computation engine (CE), that communicate using a dynamically reconfigurable interconnection network. One of the units is dedicated for instruction thread synchronization and the other for execution of the parallel threads. The main purpose of the graph engine is to enforce correct sequencing of parallel instructions based on dependencies while maintaining a high degree of parallelism among the computation threads. The computation engine can be viewed as a pool of computation processing elements (CPEs) which are allocated for executing various threads under the direction of the graph engine.

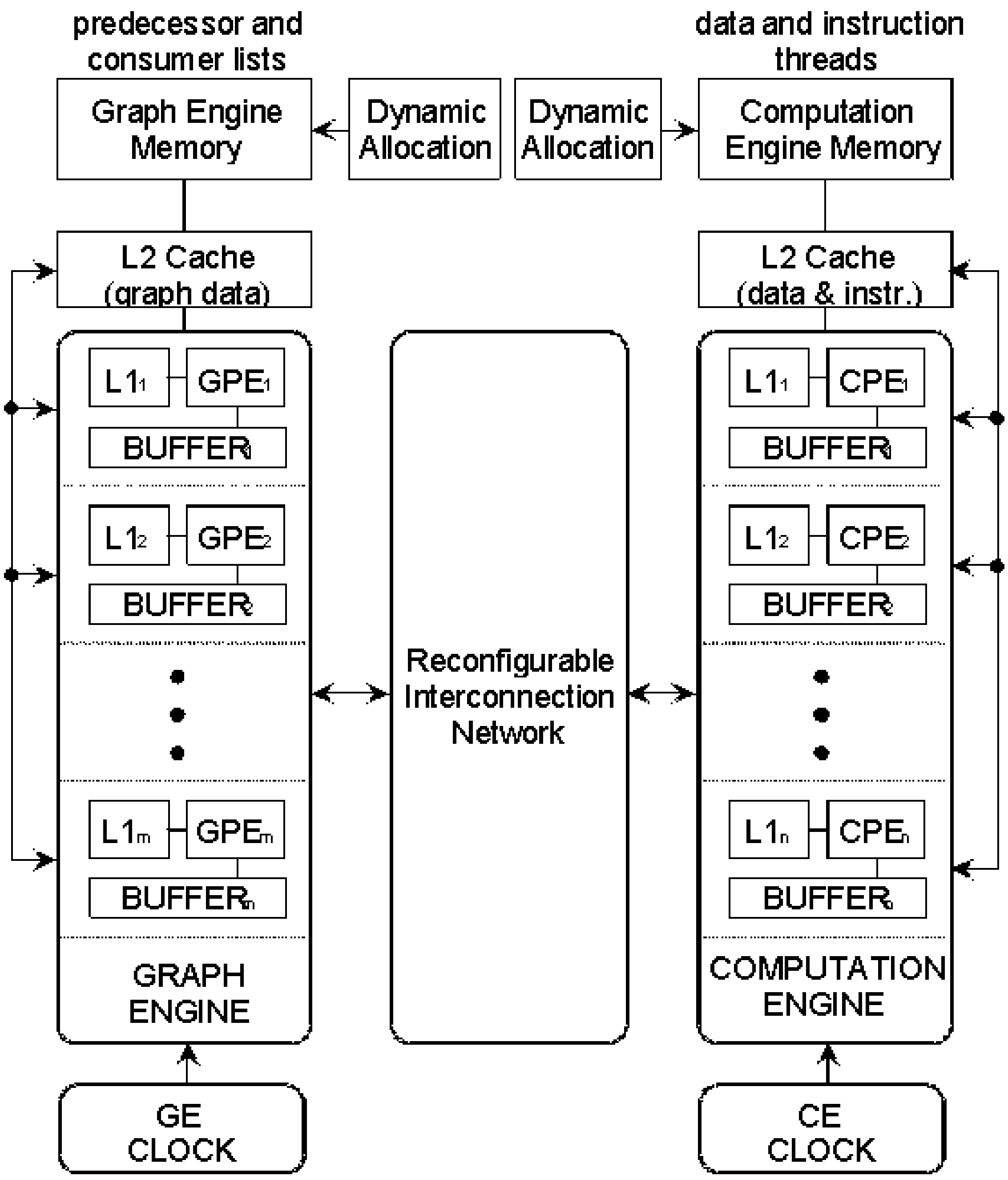


Figure 1.2.1.1. Conceptual Diagram of Proposed Architecture.

A program to be executed can be viewed as a graph with the vertices representing computation threads and the edges representing data dependencies. At program load time, information is stored in the GE memory corresponding to the program graph interconnections while the CE memory contains the collection of instruction threads and locations for storing data. During normal operation, a Graph Processing Element (GPE) reads in a node structure from the program graph memory, updates pertinent fields in the node structure, and schedules the execution of successor nodes who have all input operands available. Any processor can execute any operation ready for execution. The graph engine simply places an entry into the ready to run queue and any available processor is free to access and execute the available instruction.

The program graph which is a true data dependence graph can be generated from any language. However, to maximally exploit parallelism, a functional parallel programming language, SISAL[2], has been used to generate the data dependence graph. This graph is known as an IF1 graph.

2. Compiler Construction

Unlike other compiler construction tasks whose objective is to produce executable or object code, the IF1 compiler described here generates a data structure which is intended to be loaded into the graph engine memory module of a decoupled, multithreaded computer. Figure 2.1 shows the main block diagram of the overall program.

The front end of the IF1 compiler consists of a lexical analyzer and a parser. The lexical analyzer simply generates tokens from the IF1 source code and passes them to the parser which generates a parse tree or *Graph* data structure.

The back end consists of the code generator, IF1 viewer, and simulator. The IF1 viewer displays the IF1 graph. The code generator reads the *Graph* data structure and generates executable data structure. The simulator then simulates the multithreaded computer to estimate the run time parameters and approximate the parallelism in a program.

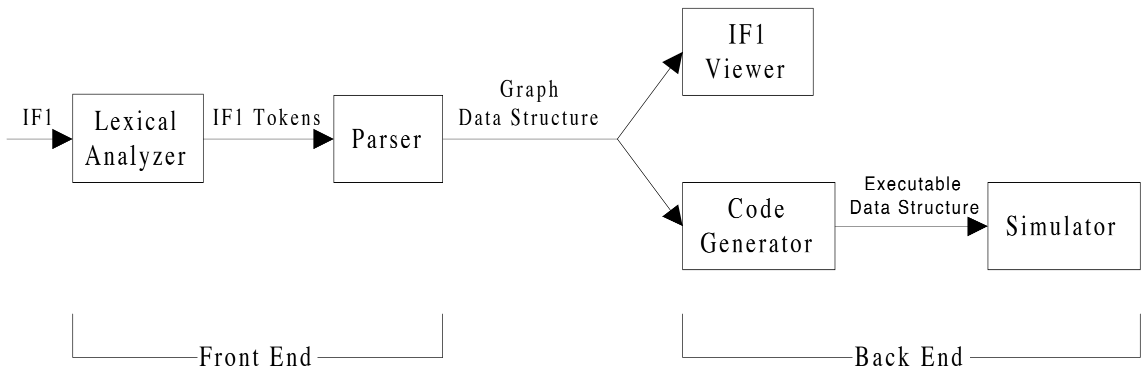


Figure 2.1. Block Diagram of IF1 Parser And Viewer

After a discussion of the IF1 language, each of the components in the block diagram in Figure 2.1 will be described.

2.1 Description of IF1 Code and IF1 Graph

IF1 is a language that describes directed acyclic graphs. There are four components to the graph: nodes, edges, types and graph boundaries. Nodes denote operations, such as *add*, *divide*, and many others as listed in Appendix A of the IF1 manual[1]. Edges represent values or data that are passed from node to node, and types can be attached to each edge or function. Graph boundaries surround groups of nodes and edges. A node can be executed as soon as all of its inputs are available. For example, a graph that represents "(a + b) / 2" is shown in Figure 2.1.1.

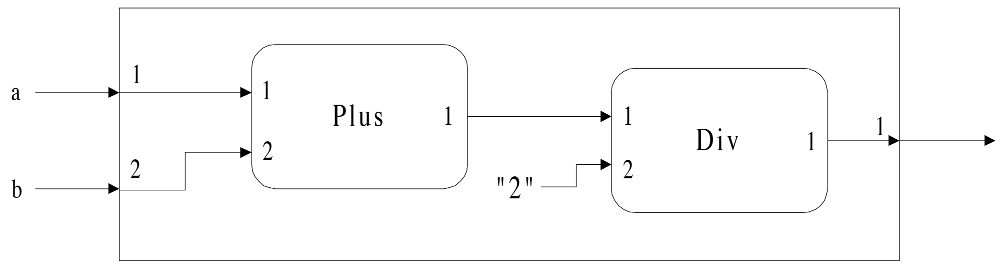


Figure 2.1.1. Graph of "(a + b) / 2"

The smaller boxes represent nodes. Currently, there are over fifty nodes defined in IF1. Both of the above nodes, or operations, require two input values and return one result. In general, the number of inputs and outputs vary according to the operation. The numbers inside the graph nodes indicate port numbers, which are used to distinguish multiple inputs and outputs. The arrows denote edges which represent data paths between nodes (or between nodes and graph boundaries). The edges also carry type information, which is not shown in the picture. A special type of edge is used to describe literal constants. The notation “2” in Figure 2.1.1, is an example of a literal constant. Types can be specified as user-defined, or, by using the built-in types. Comments may be used for any purpose.

IF1 files comprise a number of lines that contain only printable ASCII characters, and are delimited by *newline* characters. The first non-blank (non-tab) character on the line distinguishes one component from the others. The following shows the IF1 code of the graph “(a + b) / 2” shown in Figure 2.1.1.

```
C       Average
T   1    Basic     Integer
X   0   “main”
N   1    Plus
N   2    Div
E   0     1           1        1    1
E   0     2           1        2    1
E   1     1           2        1
L   2     2           1       “2”
E   2     1           0        1    1
```

To be consistent with the convention of Sisal-generated IF1 code, a program must have at most one main graph (global function denoted by token ‘X’). There can be any number of local graphs (local function denoted by token ‘G’) and external graphs (imported function denoted by token ‘I’). Some tokens may be represented by an integer number. For example, *Basic*, *Integer*, *Plus*, and *Div*, can be replaced with integer values 1, 3, 141, and 122, respectively. A line starting with token ‘C’ is a comment and anything that follows this character and precedes the *newline* is ignored. Note that comments generated by a Sisal compiler may contain some useful information.

To simplify and speed up the parsing stage, a sorting process is included. The sorting routine will move all the *type* tokens to the beginning of the file. *Type* is not dependent on scope or which subgraph it is used for. Some *types* are derived from another *type*. For such *types*, they have to follow the *type* they are derived from. The sorting process also moves all *nodes* or *compound nodes* within the subgraph to the beginning of the subgraph, and the *edges* and *literals* to the end of the subgraph. Note that a *compound node* itself is another subgraph.

A subgraph can be a function or a compound node. As a function, it will be called within another subgraph via a *Call* node. The following 5 cases are the compound nodes implemented in the IF1 compiler:

1) Select

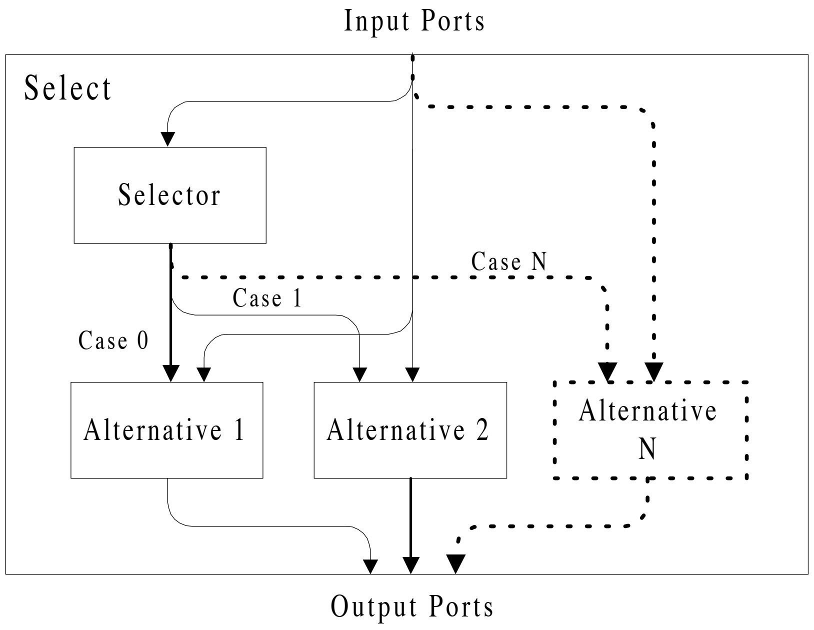


Figure 2.1.2. Select Compound Node

A *Select* compound node is used to implement a multi-way selection such as *if-then-else* or *switch-case* expressions. The output of the *Selector* subgraph determines which

Alternative subgraph is to be executed. Only one of all the possible alternative subgraphs' output edges will connect to the output of the *Select* graph.

2) Tag Case

The *Tag Case* compound node is not used very often. In Fact, of all 24 Livermore Loops[5], none of them use *Tag Case*. Thus the IF1 compiler does not generate a data structure for this compound node since its function is currently undefined for the target computer architecture. However, the IF1 viewer still recognizes this compound node.

3) ForAll

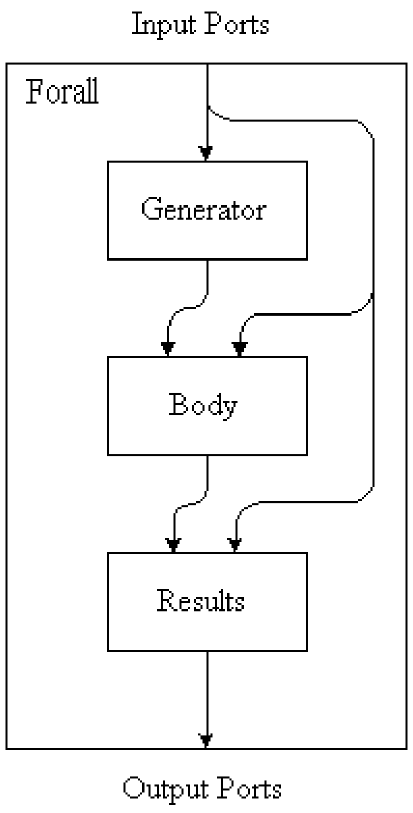


Figure 2.1.3. ForAll Compound Node

A *ForAll* compound node is used to denote independent execution of multiple instances of an expression. The *Generator* subgraph determines how many instances of the *body* subgraph are to be created.

4) LoopA

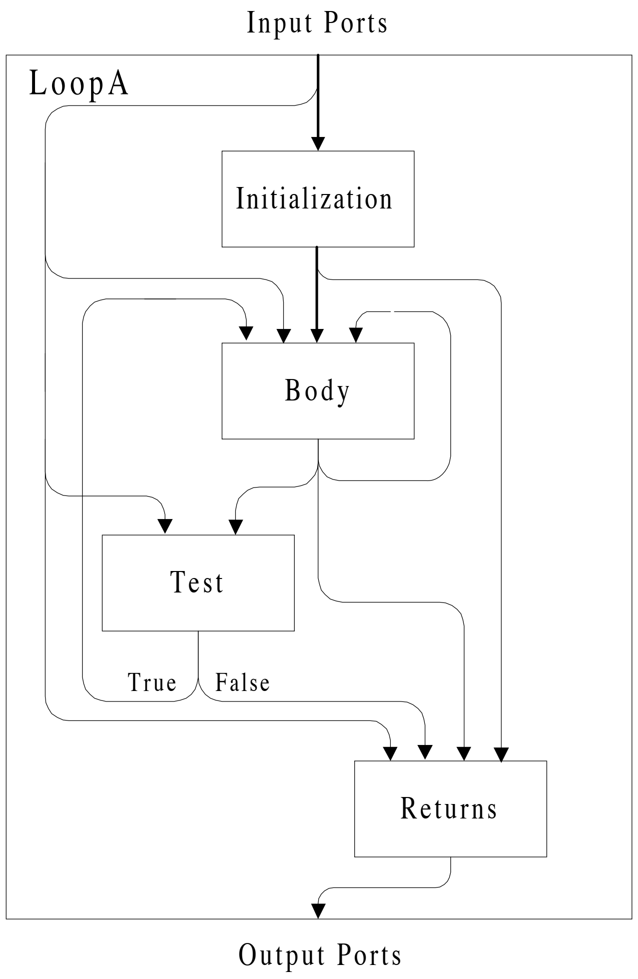


Figure 2.1.4. LoopA Compound Node

A *LoopA* compound node is used to implement iterative execution of an expression. It is similar to the *do-while* expression in the C language, in that the *Body* subgraph must execute at least once. The iteration is terminated when the output of *Test* subgraph (Boolean value) is false.

5) LoopB

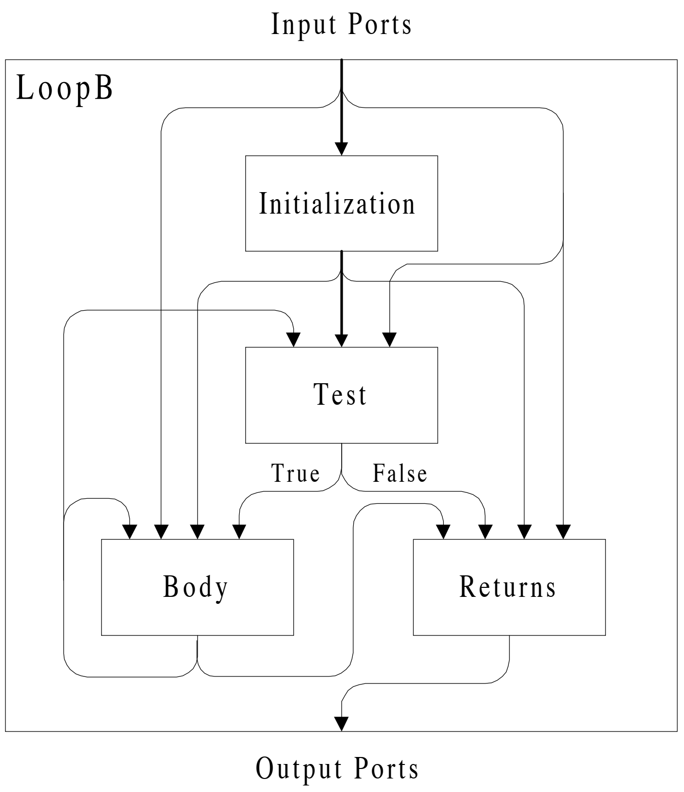


Figure 2.1.5. LoopB Compound Node

A *LoopB* compound node is used to implement an iterative execution of an expression. It is similar to the *for(;;)* expression in the C language, where the *Test* subgraph is evaluated as soon as *Initialization* finishes. If the result of the test is false, the *Return* subgraph is executed next. If the result of the test is true, the *Body* subgraph will be executed and continue on until the subsequent test result is false. With this type of node, it is possible that the body may never execute.

2.2 Compiler Front End

To build a compiler, one can use a tool such as YACC (Yet Another Compiler Compiler) which generates a skeleton of the target compiler in the C language. However, in order to do this, one must know the complete BNF rules of the language. Due to inadequate documentation, some errors and incomplete BNF rules on the existing IF1 manual[1], and no prior experience in IF1 language, the author decided to build the skeleton manually while learning the IF1 language. The skeleton still follows the YACC's style so that one can easily modify the program if needed.

Parsing refers to the process of having a program convert phrases in the language into internal structures that can be easily processed by the code generator, which in this case is the *IF1 viewer* and the *executable data structure* generator[8].

The IF1 compiler front end performs the following tasks:

1. Identify tokens (lexical analyzer).

   The complete token definition of IF1 language is given in [1]. Not all features of the IF1 language specification are recognized by the IF1 compiler; fibre formatted literal types are not currently implemented since not all versions of IF1 are able to process them[1]. Comments are not needed for the next stage, so they are ignored.

2. Parse each line according to BNF of IF1.

   The somewhat complete BNF rules of IF1 language can be found in Appendix A of the IF1 manual[1]. The BNF rules turn out to be incomplete. Minor errors were also found. Fortunately, there is Sisal compiler that generates presumably correct IF1 code, thus, the author ran several test programs and corrected or modified the BNF rules based on the generated IF1 code.

3. Sort components.

   This was explained earlier. The sorting phase simplified the parser operation.

4. Build the Symbol Table

   There are two symbol tables: type and node symbol tables.

   - The *Type symbol table* is very simple. Every type definition must have a unique label no matter where it is defined. The graph boundary is ignored here.
   - The Node symbol table is slightly more complex. All node type definitions must have unique label within the graph boundary. There may be another graph, a subgraph, within a graph with the same label as one used in another scope level. As far as the node labeling task is concerned, each subgraph is independent of another subgraph.

5. Build Parse Tree.

   The parse tree generated is a *Graph* data structure which is described in great detail in Appendix A. This is not the *executable data structure*.

The result of the above processing steps is a *Graph* data structure The *Graph* data structure is in the form of a linked list that contains all pertinent information about the node, pointers to the successor nodes, and pointers to the predecessor nodes. All information needed to reconstruct the IF1 graph can be found in this linked list. The

Graph data structure also contains all the information to be used by the compiler back end, IF1 viewer and the code generator.

2.3 Compiler Back End

In this section, we will discuss the Code Generator part in Figure 2.1. The objective of the compiler is to generate an executable data structure that can be understood by the Multithreaded Multiprocessors project[3][4]. As discussed earlier, the executable program is constructed based upon IF1 code generated by a high level language compiler. The executable program consists of a linked list of executable nodes. Each executable node includes the operation to be performed, and state information of the next node to be scheduled for execution upon completion. All this information is stored in a data structure called a *Node Template*. In Figure 2.3.1, the primary fields in the structure and the type of the field are shown. As the architecture matures, more fields may be required.

Node Template structure	Type
Node_Name	*pointer to subroutine*
Pred_Init	*integer*
Pred_Num	*integer*
Pred_Ptr	*array of integer*
Succ_Num	*integer*
Succ_Ptr	*array of pointer to Node Template structure*

Figure 2.3.1. Node Template Structure

Node_Name	identifies the type of operation, such as: *Add*, *Sub*, *Div*, etc. It contains the address of the subroutine in computation memory required to perform the actual operation.
Pred_Init	indicates number of predecessor node templates or simply the number of input operands. If all the input operands are available, then this node template is ready to be scheduled for execution. The predecessor node template must execute before this node template can be scheduled for execution.
Pred_Num	indicates the number of instances of this node template that can be executed in parallel. This field may be updated during run time when the *ForAll* compound node is encountered.
Pred_Ptr	is an array where each of its elements holds a counter associated with each instance of this node template. This array is allocated dynamically during run time and the number of elements to be allocated is *Pred_Num*. Initially, each element is assigned a value of *Pred_Init*.
Succ_Num	indicates the number of successor node templates. The successor node template needs the result of this operation before it can be scheduled for execution. The Successor nodes execute after this node template.
Succ_Ptr	is an array where each of element holds a pointer to the successor node templates. There are *Succ_Num* number of elements in the array.

The values of all the fields are known during compiler time, except for *Pred_Num* and *Pred_Ptr* which are computed during run time.

To build the executable data structure, the following steps are taken by the back end of the compiler resulting in the data structure.

1) The intermediate graph is traversed.

2a) During the interval, create a node template every time a node or a literal is found and update its predecessor node and successor node pointers.

2b) If a compound node is found, expand the node; in other words, go to step 1, starting from the beginning of each subgraph. When all subgraphs are built, relate all inputs and outputs according to the implied dependency rules of the compound node.

2c) If a function is called, expand the graph of the function.

The steps (2a and 2b) can go back to step 1. This is implemented as recursive function call. At some point within a subgraph there should be a terminal condition, otherwise, a stack overflow will result. Currently, this program does not support recursive function calls in an IF1 program since it is based on a graph traversal.

The final result is a linked list that starts from the main function. This linked list will be used by the IF1 viewer and the simulator which are discussed in Sections 3 and 4.

2.4 IF1 Example Program Graph

Figure 2.4.1 shows how to represent the expression "(a + b) / (-c)" in an IF1 graph. The corresponding linked list of node templates, or the executable structure, is shown in Figure 2.4.2.

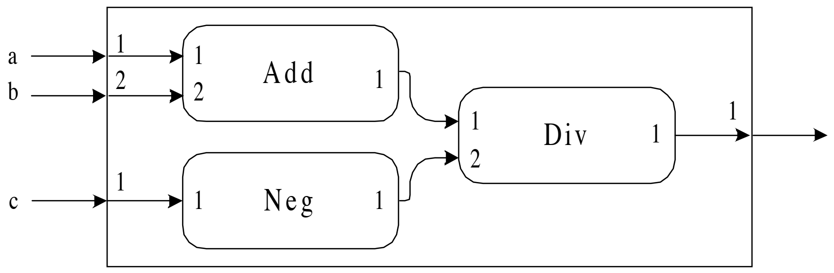


Figure 2.4.1. IF1 Graph of "(a + b) / (-c)"

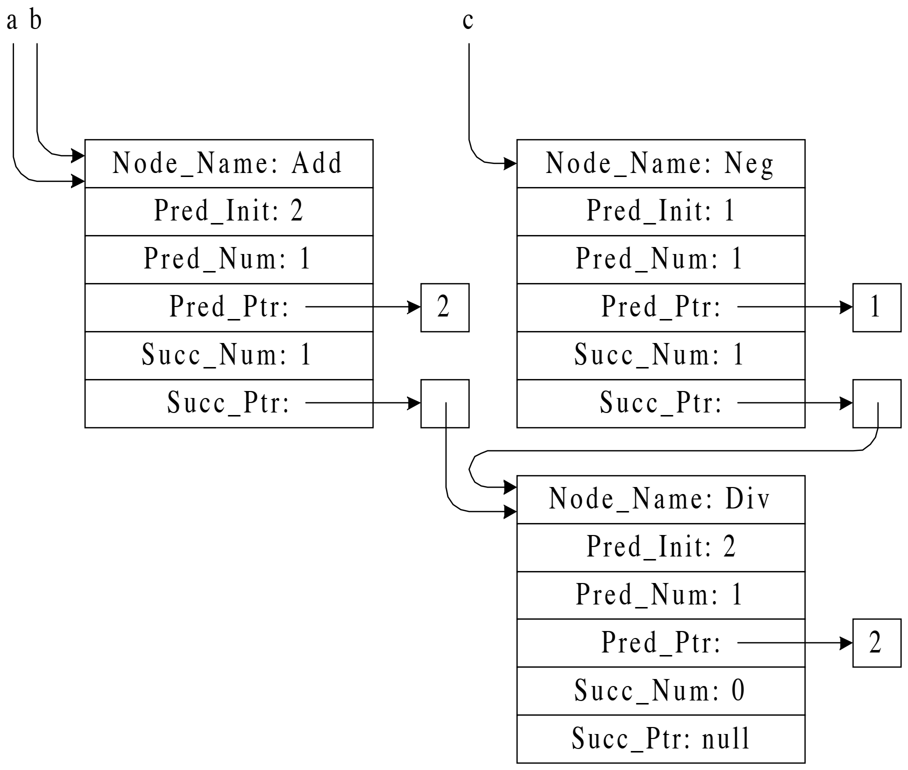


Figure 2.4.2. Executable Structure of "(a + b) / (-c)"

The *Add* node requires two operands before it can be scheduled for execution. The *Neg* node needs only one operand. The *Div* node also requires two operands before it can execute. Following the flow of the linked list, it is seen that the *Div* node must wait for the *Add* and *Neg* nodes to finish before it can be scheduled for execution. Alternatively, the *Add* and *Neg* nodes are not linked to each other, thus they can execute in parallel.

The purpose of the fields *Pred_Num* and *Pred_Ptr* is better illustrated in an executable program which contains the *ForAll* instruction.

A *ForAll* compound node is very important for the exploitation of parallelism in a program. The following discussion refers to Figure 2.1.3 which shows the internal subgraphs in a *ForAll* node and the implied dependencies among the subgraphs. *ForAll* compound nodes consist of three subgraphs: *generator*, *body*, and *result*. The *generator* duplicates the *body* subgraph N times according to the computed range value. Assuming unlimited resources, all the duplicated subgraphs can be executed in parallel. The outputs of these subgraphs are then input to the *result* subgraph which in turn outputs the final result. A *ForAll* compound node is commonly found in processing an aggregated data type such as an array. For example, incrementing the value of each element in an array can be done in parallel since each element is independent of the other.

The bounds of a *ForAll* node are computed during run time, thus, the fields *Pred_Num* and *Pred_Ptr* can only be updated then. The reason for including these two fields is to efficiently use the program graph memory which holds the node template linked list. Note that individual node template structures in each node contained in the body of the *ForAll* node are not duplicated, instead, only one node template structure is created and its field *Pred_Ptr* is expanded to **N** elements, where **N** is the number of instances.

The following example, Loop3, is taken from one of the Livermore Loops[5]. Loop3 calculates the inner product of **X** and **Z**. Loop3 takes 3 inputs **n**, **X**, and **Z**. **n** is the number of elements in the arrays **X** and **Z**. It is clear that each multiplication operation is independent of the next or previous one. Thus, a *ForAll* compound node is a good choice here.

```
type double = double_real;
type OneD   = array[double];

function Loop3( n:integer; X,Z:OneD returns double )
  for i in 1,n
      Q := X[i] * Z[i]
  returns value of sum Q
  end for
end function
```

During normal operation, the first-level nodes are read from the program graph memory. There may be more than one first-level node. The example in Figure 2.4.4 has two first-level nodes, the *Add* and *Neg* nodes. These nodes are scheduled for execution initially. Upon completion, we decrement the *Pred_Init* counts of all the successor nodes of the just executed node. If the successor's *Pred_Init* count is decremented to zero, this signifies that all data are now available for the successor node to begin execution. If the count is not zero, this signifies that the successor node still has inputs pending and should not be scheduled for execution. In the case of a *ForAll* node, each element in successor's *PredPtr* array is decremented only if that particular instance of the predecessor has completed execution.

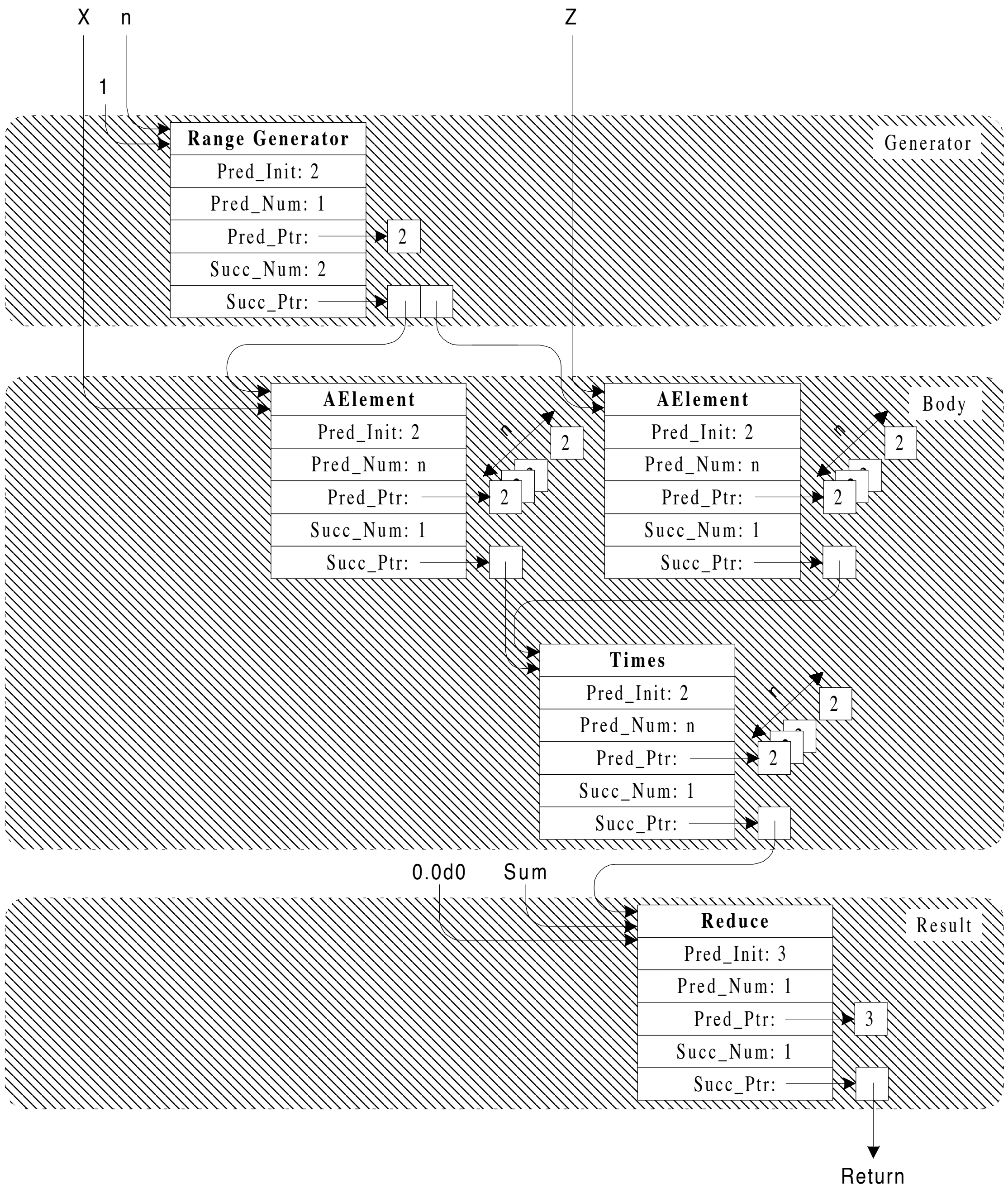


Figure 2.4.4. Node Template Linked List of Loop 3

3. Graphical Interface

A graphical display of the program graph was developed as a tool for further architecture and compiler development efforts. Although there already exists an IF1 browser, this browser is written in *HyperCard* and only runs on a Macintosh[2] computer. Since the primary project development is on a Unix system, it is very inconvenient to have to use two different systems.

The graphical interface written for Microsoft Windows utilized C++ with the help of the MFC library from Microsoft. The compiler front end part is written in C so that it can be easily ported to an X Windows system. The graphical interface to the X Windows system uses the Xt toolkit and the Athena Widget which are available for free on virtually all X Windows systems. The look and feel may vary depending on the particular window manager being used.

Figures 3.1 and 3.2 show screen views of the two different versions of the IF1 Viewer. The MS Windows version in Figure 3.1 has added menu support. The window on the left is the primary window which will appear when the program runs. The window on the right will appear when the user double clicks (or single click on the X Windows version) one of the elements in the graph list. The right side window displays the IF1 graph of the selected function, in this example, the *main* function. The input file is defined via the File Menu on the MS Windows version. On the X Windows version, user must specify the input file in the command line.

[2] Macintosh is a registered trademark of Apple Computer.

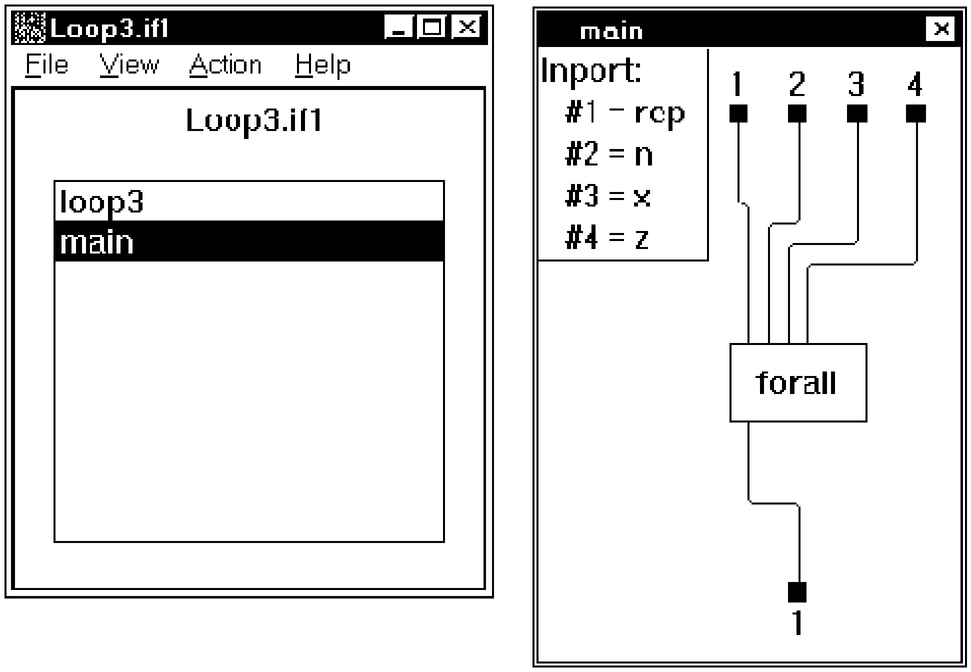


Figure 3.1. MS Windows Version of IF1 Viewer

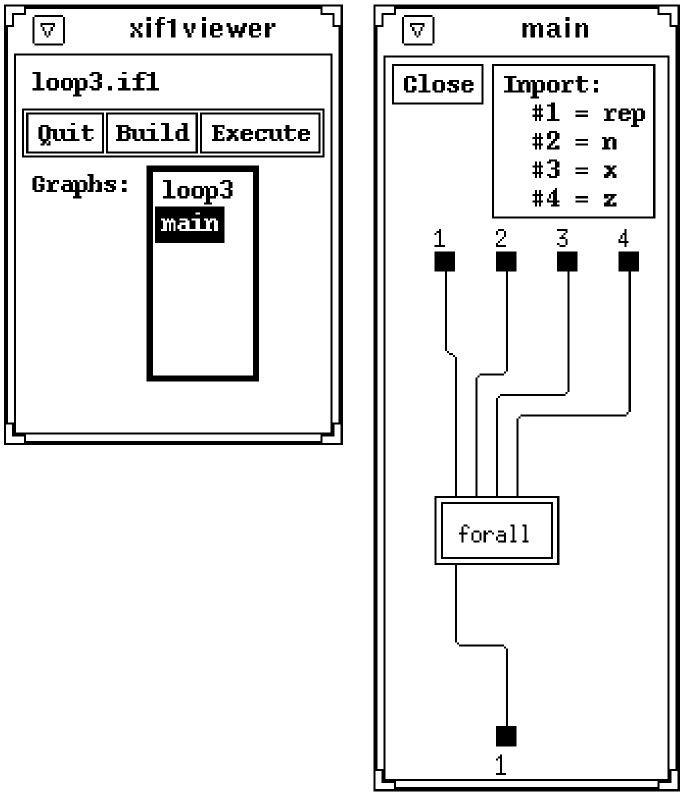


Figure 3.2. X Windows Version of IF1 Viewer

The graph list contains a list of all functions defined in the IF1 code. In this example, there are only two functions for Loop3.if1. The upper box in the right window defines the variable name associated with each input or output port. The variable names are read from the IF1 code which may include special comments that specify the variable name. If such comments are not found, the box will not appear.

As discussed in the previous section, a graph may contain one or more subgraphs and the subgraph itself may contain more subgraphs. If a node appears grayed or double-framed, it indicates that it contains a subgraph. If it is clicked, one or more subgraphs will appear. If the node is a *call* node to another function, only one subgraph will appear. If the node is a compound node such as *ForAll*, *Select*, *LoopA*, *LoopB*, or *TagCase, n* subgraphs will appear where *n* depends on the property of the node. For example, for a *ForAll* node, there will be three subgraphs: *Range Generator*, *Body*, and *Return* subgraphs.

Once an item in the Graph List (in the main window) is clicked or double-clicked, a pointer to its corresponding *Graph* structure is passed on to the callback function. This *Graph* structure is generated earlier by the compiler front end. *Graph* structure details are included in Appendix A.

Before the graph is laid out, we need to determine the level of each node which is done by traversing the subgraph. This traversing process determines the level of nodes within a subgraph. A node that depends only on input ports is a level-one node. A node whose inputs are all literals is also a level-one node. A node that depends on the output of n-level node is an (n + 1) level node. If the node depends on several nodes from different level, then it is an (n_j + 1) level node, where n_j is the largest level of all predecessor nodes' levels. Figure 3.3 shows an example of an IF1 graph with the nodes and their corresponding levels.

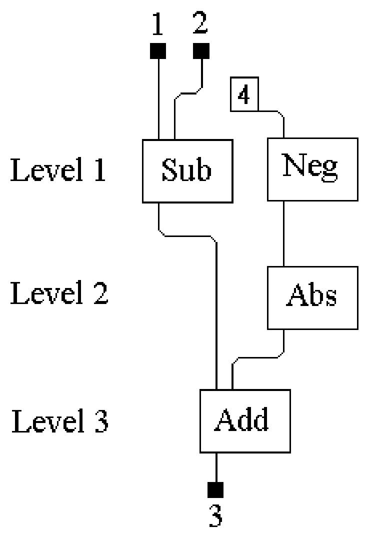


Figure 3.3. Determining Node Level in IF1 Graph.

There are 5 main steps taken to lay out the nodes and the edges on a window which are illustrated in Figure 3.4.

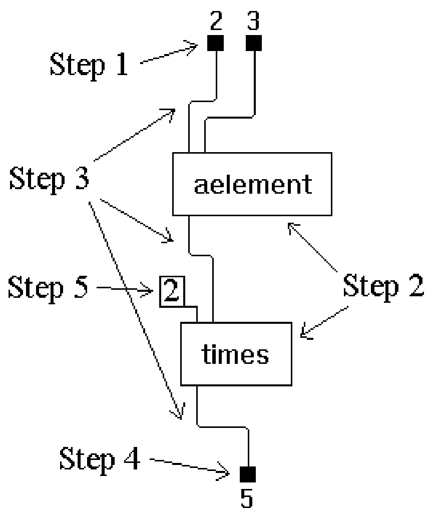


Figure 3.3. Steps Showing How The Graph Is Constructed.

1) Arrange all the input ports on the window.
2) Arrange all the nodes in such a way that the first-level nodes come first and the second-level nodes come next and so on. A compound node is identified by gray color in MS Windows or double framed in X Windows.
3) Position all the output ports after the last level nodes.
4) Draw all the edges between input or output ports and the nodes.
5) Position all the literals close to the pointed node and draw the edges.

Special care is taken to reduce number of overlapping lines. It is not uncommon to have a much larger IF1 graph than the one shown in Figure 2.4. In this case, the user needs to resize the window or grab the picture and move it up and down to see obscured portion of the graph.

4. Parallelism Analysis

This section describes the simulation capability and program. This program can be used to estimate the parallelism in an IF1 program and to generate other information that is of concern.

4.1 Methodology

The simulator does not actually simulate the running program. It simply does a graph traversal on the executable data structure and decides which node can be executed at every clock cycle. It is important to note that the simulator does not execute the node, it simply decides if a node can be executed. The number of nodes ready for execution can be more than one. This decision is based on the rules described below. The number of nodes that can be executed in parallel is called "Parallelism." Another result that is of interest is the total number of clock cycles needed to execute the program. For simplicity, the following assumptions are taken:

1) There are an unlimited number of resources (memory and processors).
2) Every simple node is executed in one clock cycle (including Noop).
3) All the compound nodes and internal functions are replaced by their corresponding subgraph.
4) Memory latency is ignored.
5) Communication delay between any two nodes is 0.

The following rules are used to decide if a node can be executed and must be followed at all times:

1) A node can be scheduled for execution if and only if all the input operands are available. This is the paradigm of the dataflow model of computation.
2) A literal is readily available at any time.

Figure 4.1.1 shows the flowchart of the simulator. This chart has been simplified greatly. In the case where there is a loop, additional steps must be taken which include restoring *Pred_Init* of all nodes in the body subgraph.

The *Execution* pool holds all the nodes that are being executed and the *Pending* pools hold the nodes that are to be executed if the above rules are satisfied. The simulation is complete when the outputs of the program are reached.

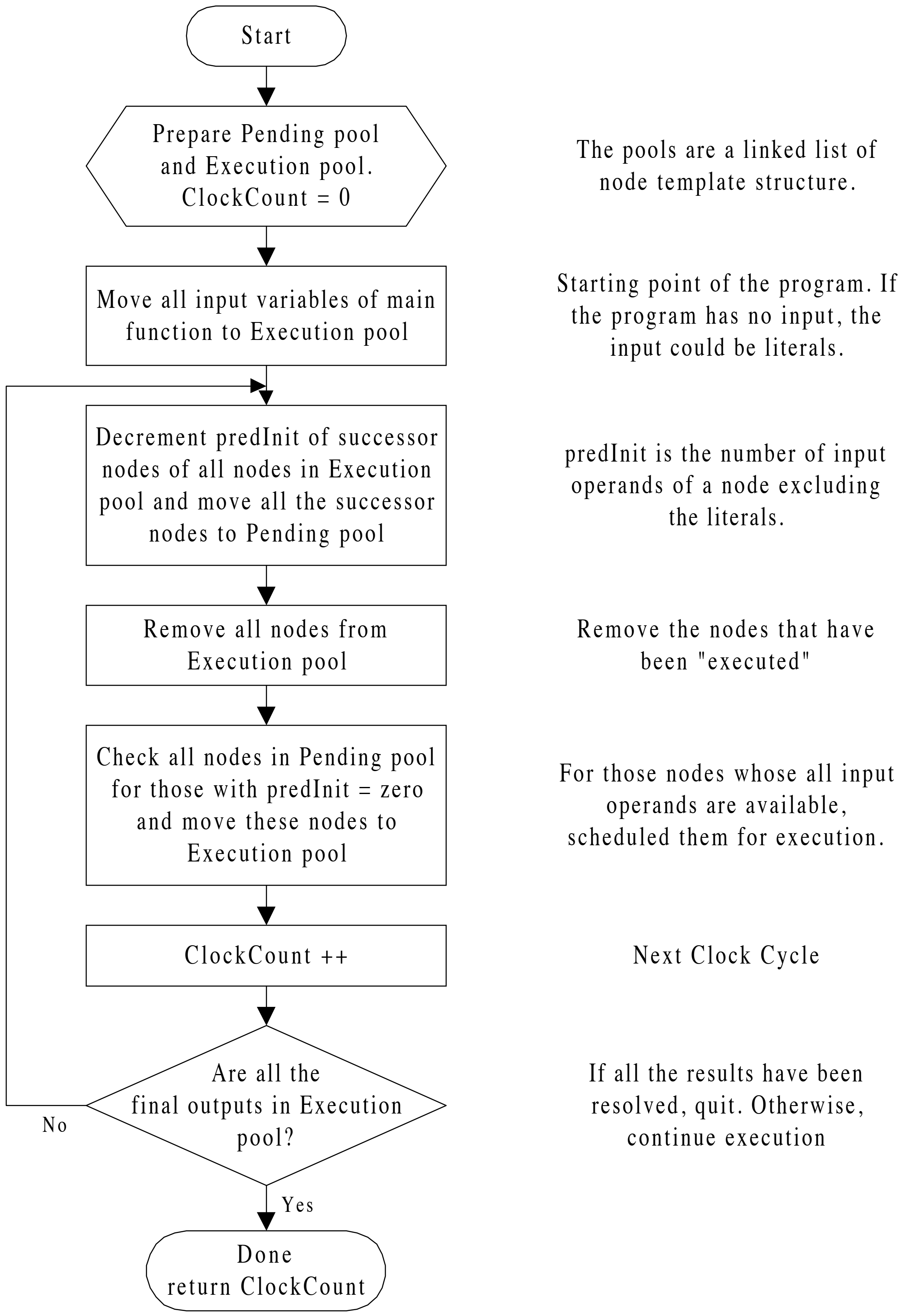
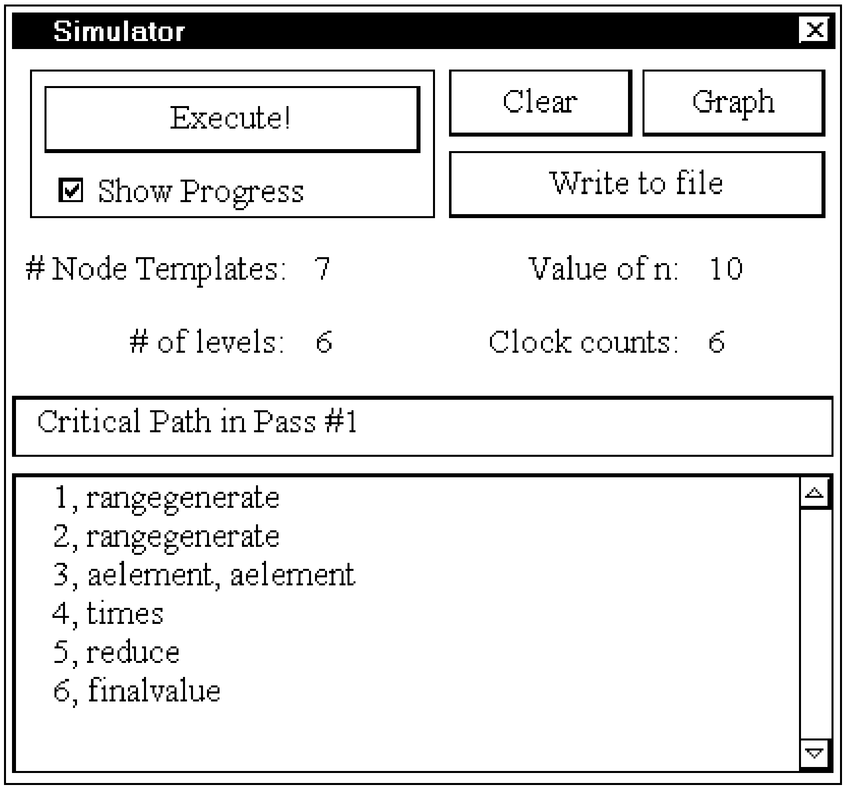

Start
Prepare Pending pool and Execution pool. ClockCount = 0
The pools are a linked list of node template structure.
Move all input variables of main function to Execution pool
Starting point of the program. If the program has no input, the input could be literals.
Decrement predInit of successor nodes of all nodes in Execution pool and move all the successor nodes to Pending pool
predInit is the number of input operands of a node excluding the literals.
Remove all nodes from Execution pool
Remove the nodes that have been "executed"
Check all nodes in Pending pool for those with predInit = zero and move these nodes to Execution pool
For those nodes whose all input operands are available, scheduled them for execution.
ClockCount ++
Next Clock Cycle
Are all the final outputs in Execution pool?
If all the results have been resolved, quit. Otherwise, continue execution
No
Yes
Done return ClockCount

Figure 4.1.1. Flowchart of The Simulator

To run the simulator, it is necessary to build the executable data structure or the so-called Node Template Structure. To build, select the *Build* menu item under the *Action* main menu and you will be asked for the *value of N*, where **N** is the number of iterations in every loop. The loop includes ForAll, LoopA, and LoopB compound nodes. In the case of LoopA and LoopB compound nodes, the body subgraph is executed **N** times. In the case of ForAll compound node, there are **N** body subgraphs to be executed in parallel. The reason there is a need for **N** is that the simulator cannot figure out when to terminate a loop since it does not actually execute the node. Thus, the result will not be the same as the actual machine simulation which executes the node and decides when to terminate a loop based on the result of execution.



Node Templates: 7
Value of n: 10
# of levels: 6
Clock counts: 6

Critical Path in Pass #1

```
1, rangegenerate
2, rangegenerate
3, aelement, aelement
4, times
5, reduce
6, finalvalue
```

Figure 4.1.2. Simulator Window

Once the Node Template Structure has been built, the window in Figure 4.1.2 will appear. The structure can then be simulated by clicking the *Execute* button. To see what nodes are being executed at every clock cycle, check the *Show Progress* check box, and the list box will list the executed nodes.

Some programs may have *Select* compound nodes which implies only one path will be taken out of **k** paths. For the *if-then-else* equivalent statement, **k** = 2, because there are only two paths to be chosen. Again, because the simulator does not compute the data value to decide which path to be taken, decision is based on which path is part of the critical path. If there are **j** number of Select compound nodes, there are $\mathbf{2^j}$ paths to be decided; thus, the execution will take $\mathbf{2^j}$ passes before it can decide which path is the longest path, thus, the ciritcal path. There could be more than one such path; in this case, only the first one is chosen. The pass number will be announced above the list box.

Once the simulation has completed, the Parallelism graph can be viewed by clicking the *Graph* button or it can be saved as a text file by clicking the *Write to file* button. The text file has the following format:

Clock #	Parallelism	# of incoming arcs	# of outgoing arcs

The first column is the clock number, the second column is the parallelism at that clock cycle, the third column is the total number of incoming arcs at that clock cycle, and, the fourth column contains the total number of outgoing arcs at that clock cycle.

For the X-Windows version, click on the *Build* button to build the structure. The value of **N** is by default 10 but it can be changed via the command line. For example:

```
xif1viewer loop1.if1  25
```

The above command will execute the if1viewer with the IF1 program **loop1.if1** and **N** value of **25**. When building is complete, click on *Execute* button to execute. The result will be stored to a text file which has the same format as that of MS Windows version.

4.2 Results

The Livermore Loops were used to test the simulator. The Livermore Loops are FORTRAN loops from actual production codes that run at Lawrence Livermore National Laboratory. They represent the type of computation kernels typically found in large-scale scientific computing. They range from common mathematical operations, such as inner product and matrix multiplication to searching and sorting algorithms. The loops provide an excellent test bed to evaluate the appropriateness and expressive power of parallel languages and architectures. The Sisal source codes of all the 24 Livermore loops are listed in Appendix C.

Table 4.2.1. shows the value of **N** chosen for each loop. Any other value of **N** can be used, however, it must be taken into account that some programs may have nested loops which will drastically increase the size of the output and the parallelism graph may look very packed as is shown in some of the results.

Loop Name	Value of **N**	# of Levels	# of Node Templates
Loop 1	990	9	13
Loop 2	3	99	27
Loop 3	1001	4	5
Loop 4	35	16	109
Loop 5	25	103	8
Loop 6	10	93	15
Loop 7	995	13	34

Loop 8	100	16	119
Loop 9	101	15	40
Loop 10	101	15	50
Loop 11	25	78	6
Loop 12	1000	6	7
Loop 13	3	141	162
Loop 15	101	24	160
Loop 19	5	74	28
Loop 20	5	132	69
Loop 21	10	12	16
Loop 22	101	13	18
Loop 24	10	62	10

Table 4.2.1.

The parallelism profiles are listed in Appendix D. There are two graphs associated with each program. The first one is the plot of the number of incoming arcs and outgoing arcs at every clock cycle. The other graph is the plot of parallelism at every clock cycle.

There is a similar simulation done by John T. Feo[6]. One similarity between the two is that both respect data and logical dependencies among nodes. However, the results of the simulation differ from the ones we have here because of the way a program is simulated. The main difference is that Feo computed the parallelism based on the result of Sisal interpreter and profiler which emulate execution of every operation based on the input data and passes the data to the next node. Thus, the loops whose number of iterations are dependent on a variable will terminate when the certain condition is met. Similarly, when decision is needed to decide which path is to be taken when Select compound node is found, the Sisal Interpreter actually computes the *Selector* subgraph and chooses the path based on the result of computation. On the other hand, our simulator iterates all loops **N** times where **N** is fixed. And when a Select compound node is found, only the path that contributes to the critical path is taken. No actual data computation is performed.

Another difference that plays an important role is that our simulator associates one node with one operation. Feo's, on the other hand, also considers the scattering and gathering latencies. This is better illustrated in a program that uses the *ForAll* compound node as shown below, in fact, only in the *ForAll* node can this difference be noted.

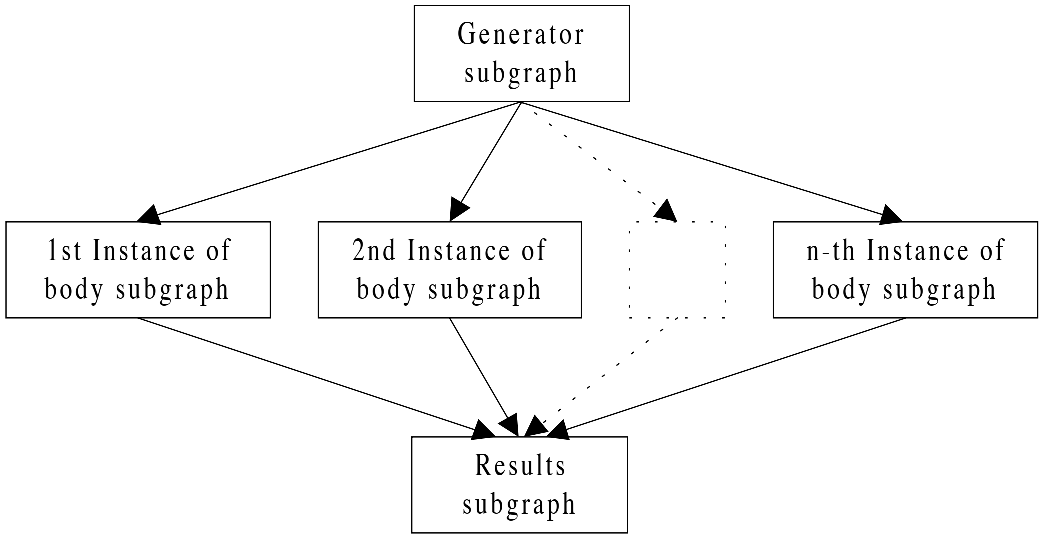

Generator subgraph
1st Instance of body subgraph
2nd Instance of body subgraph
n-th Instance of body subgraph
Results subgraph

Figure 4.2.1. ForAll Graph.

Assuming that Figure 4.2.1 is the overall IF1 graph and in each subgraph there is only one level node, then the following table shows the result based on Feo's (left) and ours (right).

Clock #	Sisal Interpreter	Our Simulator
1	N	1
2	N * # of nodes	N * # of nodes
3	N	1

At clock # 1, Feo assumes that **N** *body* subgraphs have been scattered, thus there are **N** operations. At clock # 2, assuming the depth of the body subgraphs is only one and there are **k** nodes, then there are (**N** * **k**) operations. At clock # 3, there are **N** results, which are gathered, so there are **N** operations. In contrast, our simulator only counts the total number of nodes at each clock #.

The next example is one of the Livermore Loops, **Loop 3**. This IF1 program has been stripped such that only the loop itself is present, the main function that calls the loop is removed. The IF1 graph is shown in Figure 4.2.2. The result is shown in the following table and plotted in Figure 4.2.3.

Clock #	Sisal Interpreter	Our Simulator	
1	1001	1	□ Scatter
2	2002	2002	
3	1001	1001	
4	1001	1	□ Gather

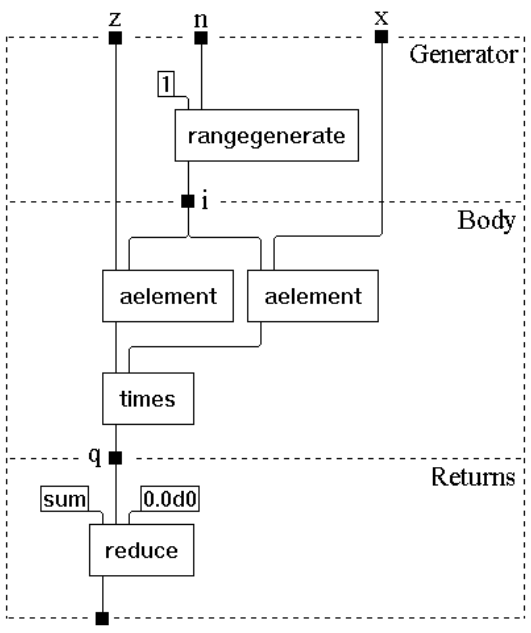
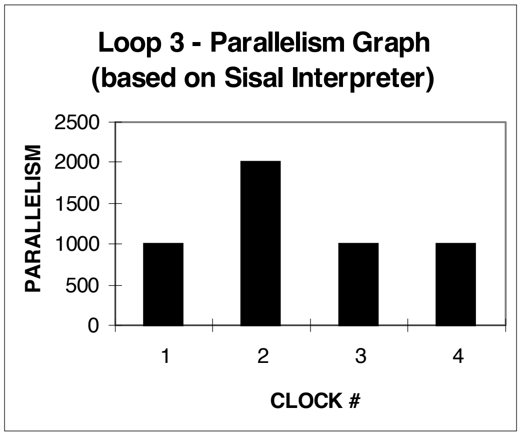
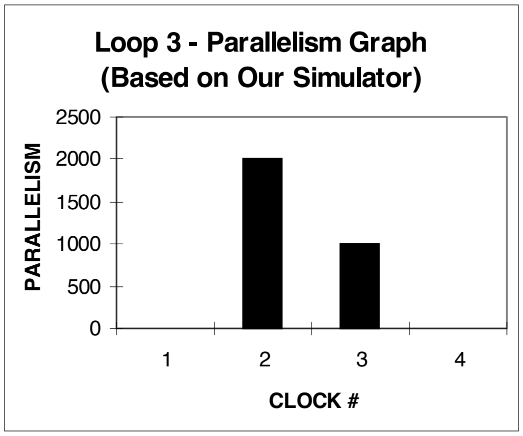


Figure 4.2.2. IF1 Graph of Loop 3

Note that the body subgraph is duplicated **N** times which is not shown here for efficiency reasons. Recall that in the executable data structure, the body subgraph is not duplicated;

instead, the field *Pred_Ptr* will grow to **N** elements and *Pred_Num* will be **N** resulting in graph memory conservation in the target architecture.





4.2.3. Parallelism Graph of Loop 3.

Figure 4.2.3 shows the plots of the parallelism based on the Sisal Interpreter and our simulator. As can be clearly seen, only prior to and after the body subgraphs are the results different because one takes into account the scattering and gathering latencies, whereas the other does not. Within the body subgraphs, however, the results are exactly the same.

Appendix D shows the graphs of all the test programs as mentioned earlier. Both parallelism and incoming or outgoing arcs are plotted for each test case. The Sisal source codes of all these 24 Livermore Loops (including several that are not simulated) are included in Appendix C. All these files and Sisal compiler are available at *http://sisal.llnl.gov/*

5. Conclusion And Future Work

The objective of this project is to create a compiler that will parse an IF1 program and generate an executable data structure to be used by a Multithreaded Parallel Processing Architecture[3][4]. This objective has been achieved successfully. In addition, several tools that are needed to support the ongoing research have also been developed. These tools are the IF1 Viewer and the Simulator. The IF1 Viewer helps researchers to see the IF1 graph visually, thus, enables them to quickly see the flow of data based on the data dependence restrictions. The simulator enables researchers to estimate the number of resources or processors required by a program given the desired input parameters. It does this by computing the parallelism in a program, thus, shows what programs are or are not rich in parallelism.

Several enhancements can be made to the compiler. Though it has successfully generated the executable data structures for all 24 Livermore Loops, it may fail to do so for other exceptional IF1 programs. This may due to new enhancement added to the IF1 language itself or the inability of IF1 compiler to recognize some tokens that are not currently supported. The IF1 compiler also does not support recursive function call due to the graph traversal algorithm used. One thing that can certainly be improved is the user interface. Though it plays little role in the whole project, it eases the interaction between the user and the program. Lastly, new optimization methods have been identified that should be incorporated into the compiler back end[7].

Future work is still needed to fully utilize the result generated by IF1 compiler. There is a need to convert the executable data structure into graph engine memory image so that graph processing elements can efficiently read the graph information and process it accordingly. There is also a need for a template that associates a node with its corresponding assembly instructions. Processors with the same architecture or assembly instructions can use a template created solely for them, and other processors with different architecture can use different template created for them. Thus, it allows various processors to exist and work together in one system.

Finally, more work may be needed as the architecture matures. The program has been written and categorized into several files to allow other programmers to quickly find and modify parts of the program as needed. Some platform independent codes are separated into different files. It is the intention of the author to ease any future enhancement to the program.

Bibliography

[1] IF1 An Intermediate Form for Applicative Languages, Ver. 1.0 M-170, 146 University of California Davis, July 31, 1985.

[2] SISAL: Stream and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2, M-146 University of California Davis, March 1, 1985 .

[3] Performance Evaluation of a Data Driven Architecture, Mitch Thornton, Proceedings of the 1996 Arkansas Computer Conference, March 1996, pp. 71-76.

[4] A Decoupled Graph/Computation Data-Driven Architecture with Variable-Resolution Actors, Paraskevas Evripidou and Jean-Luc Gaudiot, *Proceedings of the 1990 International Conference on Parallel Processing*.

[5] The Livermore Loops in Sisal, John T. Feo, Lawrence Livermore National Laboratory, August 26, 1987.

[6] An Analysis of the Computational and Parallel Complexity of the Livermore Loops, John T. Feo, Parallel Computing, Elsevier Science Publishers 0167-8191, July 1988 pp. 163-185.

[7] Graph Analysis and Transformation Techniques for Run-Time Minimization in a Multi-Threaded Architecture, Mitch Thornton, David Andrews, Proceedings of the 1997 30th Hawaii International Conference on Systems Sciences, January 1997.

[8] The Theory And Practice of Compiler Writing, Jean-Paul Tremblay and Paul G. Sorenson, McGraw-Hill Book Company, 1985.

Appendix A
Graph And Other Structures

The following data structures are defined in file **compiler.h**. Only those structures needed to build *minimal IF1 compiler* are described here. The *minimal IF1 compile*, described in Appendix B, reads IF1 source code and generate a node template linked list.

NODETEMPLATE structure

char ***nodename**	currently points to a string that indicates the name of the node.
int **predInit**	is the number of predecessor. Only predecessors with **type** = **NT_NODE** are counted.
int **predNum**	indicates there are **predNum** elements in **predPtr** array.
Int ***predPtr**	is an array of size **predNum**. Each element has a value of **predInit**. The size is currently one element. More elements may need to be allocated during run time. The number of elements should correnpond to the number of instances created for this node.
int **succNum**	is the number of elements in **succPtr** array.
NODETEMPLATE ****succPtr**	is an array of size **succNum.** Each element is a pointer to successors.
NTTYPE **type**	indicates the type of node: **NT_NODE** for simple node; **NT_LITERAL** for literal.
int **predecNum**	is similar to **predInit** but all predecessors are counted, including literals.
NODETEMPLATE ****predecPtr**	is similar to **succPtr**, but it holds pointers to all predecessors, including literals. There are **predecNum** elements in this array.
int **level**	indicates the level of node.
int **nCase**	Indicates that this node will be executed depending on the output of predecessor node. If **nCase** = 10, then this node is executed if only if the output of predecessor node is FALSE. If **nCase** = 20, then it is executed if the output of predecessor node is TRUE. These two cases are used to determine when to stop executing body of LoopA or LoopB. If 0 < **nCase** < 10, then this node is executed

	when the output of predecessor node is **nCase**. This is used in switch-case or if-then-else statement. Currently, only 10 cases are supported.
SCOPING ***scoping**	contains the scope information of the node.
NTREL ***succType**	is an array of size **succNum.** Element no *n* indicates relation between this node and the successor no *n*,
int **depthOfScope**	indicates the depth of the scope where this node is in. Inner scope has larger **depthOfScope** than outer scope: **NTREL_DATADEP** for true data dependency; and **NTREL_IMPLIEDDEP** for implied dependency.
NODE ***n**	is a pointer to part of GRAPH data structure.

SCOPING structure

int **cnId**	is a unique ID of the compound node where this node is in.
int **gId**	is a unique ID of the graph where this node is in.
int **gNo**	indicates what subgraph in a compound node this node is in. If **gNo** = 0, then it is not in any compound node. If **gNo** = 1, then it is the first subgraph of the compound node. The first subgraph of ForAll compound node is "Range Generator", that of LoopA or LoopB compound node is "Initialization", that of Select compound node is "Selector."
int **inst**	is the number of instances to be generated for this node.
SCOPING ***link**	is a pointer to outer scope.

NODE structure

COMPONENT ***node**	points to IF1 information of the node, such as: name of node, line no, input and output port numbers.
int **noInPort**	is the number of input ports of this node.
int **noOutPort**	is the number of output ports of this node.
PORT ***outPort**	is an array of type **PORT**. If port no **k** connects to a node, then **output[k]** contains pointer to the pointed node. If it points to the output of a graph boundary, **output[k]** is NULL.
GRAPH ***graph**	is undefined if node is not a compound node. If the node is a compound node, **graph** points to a GRAPH structure of the compound node. Note that a compound node is a subgraph.
NODE ***next**	is a linked list of all nodes within the subgraph or graph boundary.
int **level**	is the level of the node within the subgraph.

NODETEMPLATE ***nt**	points to the corresponding node template structure.
NTYPE **type**	is the type of the node: **N_NODE** for simple node; **N_FUNCTION** for function call; and **N_CNODE** for compound node.

COMPONENT structure

TOKEN ***token**	indicates the type of token: opencurly, closecurly, quote, number, literal, etc.
int **lineNo**	indicates where this component is defined in the IF1 file.
COMPONENT ***next**	points to the next component.

PORT structure

int **portNo**	is the port number.
NODE ***node**	points to a node where this port is connected to.
PORT ***next**	is a linked list of **PORT** that points to all nodes that this port connects to. Note that a port may connect to more than one node. The end of list is reached when **next** is NULL.

GRAPH structure

COMPONENT ***graph**	is the IF1 information on this graph. Only components with first tokens **X** and **G** are represented.
int **noInPort**	is the number of input ports of this graph.
int **noOutPort**	is the number of output ports of this graph.
PORT ***inPort**	is an array of size (**noInPort + 1**) elements. Each element points to a node. If the element points to the output of the graph, the element is NULL.
GRAPH ***next**	points to the next subgraph. This is only used by compound node where there are several subgraphs in one compound node. For example, ForAll has 3 subgraphs.
NODE ***nodeLList**	is a linked list of all nodes in this subgraph.
LITERAL ***literalLList**	is s linked list of all literals in this subgraph.

LITERAL structure

NODE ***node**	points to node whose one of the inputs is this literal.
int **portNo**	indicates which input port of a node that this literal connects to.
COMPONENT ***literal**	points to the IF1 information of this literal.
LITERAL ***next**	is a linked list of all literals in this graph boundary.
NODETEMPLATE ***nt**	points to a node template associated with this literal.

Appendix B
File Description

Common Files for both MFC version and X Windows version of IF1 compiler:

compiler.h	External function and variable definitions.
token.h	List of all IF1 basic types and nodes.
keyword.h	Tables containing strings and their numeric definitions.
token.cpp	Lexical Analyzer.
parse.cpp	IF1 Parser.
sort.cpp	Component sorter.
symtab.cpp	Symbol table builder.
Build.cpp	Parse tree builder (generates Graph data structure).
nodetemp.cpp	Node template linked list builder.
Common.cpp	Common and basic functions (initialization, safe malloc, etc).
linklist.h linklist.cpp	Link list class (used by IF1 Viewer only).
ptrarray.h ptrarray.cpp	Dynamically rezisable array class.
ntclass.h ntclass.cpp	Node Template class (used by node template builder).
wiretrac.h wiretrac.cpp	Wire tracker class (used by IF1 Viewer to keep track of lines from overlapping).
object.h object.cpp	Basic class of all other classes. This is a base class for LinkList, PtrArray, NTClass, WireTracker classes.

MFC-specific Files:

if1viewr.h if1viewr.cpp	Constructor initialization (generated by Visual C++).
mainfrm.h mainfrm.cpp	The main frame of the application (generated by Visual C++).
if1vidoc.h if1vidoc.cpp	Document part of MFC's Doc-View paradigm (generated by Visual C++).
if1vivw.h if1vivw.cpp	Viewer part of MFC's Doc-View paradigm (generated by Visual C++).
stdafx.h stdafx.cpp	MFC header files.
subgraph.h subgraph.cpp	A derived dialog class that displays IF1 graph.
dispnt.h dispnt.cpp	A derived dialog class that displays Node Template linked list.

debugdia.h debugdia.cpp	A simulator dialog box.
pargraph.h pargraph.cpp	The parallelism graph viewer.
graphbtn.h graphbtn.cpp	A derived button class to indicate a node with internal graph (used to indicate compound nodes or function calls).
loopdlg.h loopdlg.cpp	A dialog box asking the user for a value of N.
resource.h	Windows resource definition.
if1viewr.rc	Windows resource file.

X Windows-specific Files:

main.cpp	Main function.
subgrapx.h subgrapx.cpp	A class that displays IF1 graph
metafile.h metafile.cpp	A metafile class used to emulate windows metafile on X windows system.
msgbox.h msgbox.cpp	A class that displays messages to user.

A minimal IF1 compiler will require these files:

compiler.h, token.h, keyword.h, token.cpp, parse.cpp, sort.cpp, symtab.cpp, Build.cpp, nodetemp.cpp, Common.cpp, ptrarray.h, ptrarray.cpp, ntclass.h, ntclass.cpp, object.h, object.cpp, and **main.cpp** file which contains the following code:

```
#include <stdio.h>
#include “compiler.h”

void main(int argc, char **argv) {
  nLoop = atoi(argv[2]); /* value of n */
  Filename = argv[1];

  InitializeVariables();  /* Initialize necessary variables */
  Parse();                /* parse IF1 */
  SortComponents();       /* sort components */
  BuildTypeSymbolTable(); /* build symbol tables */
  BuildGraphSymTab();     /* build graph symbol tables */
  BuildGraph();           /* build parse tree (Graph structure)*/
  TraverseGraph(rootGraphStructure.next);
                          /* Traversing Graph to calculate */
                          /*   level */
  BuildStructure();       /* build the Node Template linked list */
}
```

The node template linked list starts from variables **input** which is an array of pointers to **NODETEMPLATE** structure.

```
NODETEMPLATE **input;
```

There are **k** number of elements, where **k** is the number of input variables of the function. For example, Loop1 which has the following function definition:

```
function Loop1( n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
```

has 6 input variables: **n**, **Q**, **R**, **T**, **Y**, and **Z**

input[0] points to node template **n**.
input[1] points to node template **Q**.
input[2] points to node template **R**.
input[3] points to node template **T**.
input[4] points to node template **Y**.
input[5] points to node template **Z**.

input[0]->succPtr[0] points to the node that takes **n** as one of the inputs.
input[1]->succPtr[0] points to the node that takes **Q** as one of the inputs.
    *
    *
    *
input[5]->succPtr[0] points to the node that takes **Z** as one of the inputs.

The number of input variables or **k** is equal to **(mainGraph->noInPort + 1),** where **mainGraph** is of type **GRAPH*. MainGraph** points to the **Loop1** GRAPH structure.

```
GRAPH *mainGraph;
```

The linked list could have more than one endpoints (where the linked list terminates). These endpoints are reached when the traversal finds **output[m]** node templates, where **m** is the number of return variables. For example, Loop1 has only one return value, thus, the size of output is 1 element.

Output[0] points to node template **OneD**.

```
NODETEMPLATE **output;
```

Appendix C
Sisal Source Codes of The 24 Livermore Loops

```
% LOOP 1
% Hydro Fragment
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop1( n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for K in 1,n
       X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
   returns array of X
   end for
end function

function Main( rep,n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for i in 1, rep
      X := Loop1( n, Q, R, T, Y, Z );
   returns value of X
   end for
end function
```

```
% LOOP 2
% ICCG Excerpt (Incomplete Cholesky - Conjugate Gradient)
% Sequential Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop2( n:integer; V,Xin:OneD returns OneD )
  for initial
      IL    := n;
      IPNTP := 0;
      X     := Xin;
  while ( IL > 1 ) repeat
      IPNT  := old IPNTP;
      IPNTP := old IPNTP + old IL;
      IL    := old IL / 2;
      X     := for initial
                    k  := IPNT+2;
                    Xt := old X;
                    i  := IPNTP;
               while ( k <= IPNTP ) repeat
                    k  := old k + 2;
                    i  := old i + 1;
                    Xt := old Xt[i: old Xt[old k] -
                           (V[old k]   * old Xt[old k-1]) +
                           (V[old k+1] * old Xt[old k+1])];
               returns value of Xt
               end for;
  returns value of X
  end for
end function

function Main( rep,n:integer; V,Xin:OneD returns OneD )
   for initial
       i := 1;
       X := Xin;
   while ( i <= rep ) repeat
       i := old i + 1;
       X := Loop2( n, V, old X );
   returns value of X
   end for
end function
```

```
% LOOP 3
% Inner Product
```

```
Define  Main

type double = double_real;
type OneD   = array[double];

function Loop3( n:integer; X,Z:OneD returns double )
  for i in 1,n
      Q := X[i] * Z[i]
  returns value of sum Q
  end for
end function

function Main( rep,n:integer; X,Z:OneD returns double )
   for i in 1, rep
      V := Loop3( n, X,Z );
   returns value of V
   end for
end function
```

```
% LOOP 4
% Banded Linear Equations
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop4(n: integer; X, Y: OneD returns OneD )

  let
      steps := n / 5;

      T1, T2, T3 :=
          if steps < 6 then
             X[6] -    for i in 1, steps
                       returns value of sum
                           X[6 - 6 + i] * Y[5 * i]
                       end for,
             X[503] -  for i in 1, steps
                       returns value of sum
                           X[503 - 6 + i] * Y[5 * i]
                       end for,
             X[1000] - for i in 1, steps
                       returns value of sum
                           X[1000 - 6 + i] * Y[5 * i]
                       end for
          else
             ( (1.0d0 - Y[30]) *
               (X[6]  - for i in 1, 5
                        returns value of sum
                            X[6 - 6 + i] * Y[5 * i]
                         end for))
             - for i in 7, steps
               returns value of sum
                   X[6 - 6 + i] * Y[5 * i]
               end for,
             ( (1.0d0 - Y[30]) *
               (X[503] - for i in 1, 5
                         returns value of sum
                             X[503 - 6 + i] * Y[5 * i]
                          end for))
             - for i in 7, steps
               returns value of sum
                   X[503 - 6 + i] * Y[5 * i]
               end for,
             ( (1.0d0 - Y[30]) *
               (X[1000]  - for i in 1, 5
                           returns value of sum
                               X[1000 - 6 + i] * Y[5 * i]
                            end for))
             - for i in 7, steps
               returns value of sum
                   X[1000 - 6 + i] * Y[5 * i]
               end for
          end if
  in
      X[6: T1 * Y[5]; 503: T2 * Y[5]; 1000: T3 * Y[5]]
  end let
end function

function Main( rep,n:integer; Xin,Y:OneD returns OneD )
   for initial
       i := 1;
       X := Xin;
   while ( i <= rep ) repeat
       i := old i + 1;
       X := Loop4( n, old X , Y );
   returns value of X
   end for
end function
```

```
% LOOP 5
% Tri-Diangonal Elimination, Below Diagonal
% Sequential Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop5( n:integer; XIn,Y,Z: OneD returns OneD )
  for initial
      i := 2;
      X := XIn[1];
  while i <= n repeat
      i := old i + 1;
      X := Z[old i] * (Y[old i] - old X)
  returns array of X
  end for
end function

function Main( rep,n:integer; Xin,Y,Z:OneD returns OneD )
   for i in 1, rep
       X := Loop5( n, Xin, Y, Z );
   returns value of X
   end for
end function
```

```
% LOOP 6
% General Linear Recurrence Equations
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

function Loop6( n:integer; B:TwoD; Win:OneD returns OneD )
  for initial
      i := 2;
      W := Win;
  while i <= n repeat
      i := old i + 1;
      V := for k in 1, old i - 1 returns
           value of sum B[old i,k] * old W[old i - k]
           end for;
      W := old W[old i: old W[old i] + V];
  returns value of W
  end for
end function

function Main( rep,n:integer; B:TwoD; Win:OneD returns OneD )
   for initial
      i := 1;
      W := Win;
   while ( i <= rep ) repeat
      i := old i + 1;
      W := Loop6( n, B, old W );
   returns value of W
   end for
end function
```

```
% LOOP 7
% Equation of State Fragment
```

```
Define  Main

type double = double_real;
type OneD   = array[double];

function Loop7( n:integer; R,T:double; U,Y,Z: OneD; returns OneD )
  for k in 1,n returns
  array of U[k] + R * (Z[k]   + R * Y[k])
                + T * (U[k+3] + R * (U[k+2] + R * U[k+1])
                + T * (U[k+6] + R * (U[k+5] + R * U[k+4])))
  end for
end function

function Main( rep,n:integer; R,T:double; U,Y,Z: OneD; returns OneD )
   for i in 1, rep
       W := Loop7( n, R, T, U, Y, Z );
   returns value of W
   end for
end function
```

```
% LOOP 8
% A. D. I. Integration
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];
type ThreeD = array[TwoD];


function Loop8( n:integer; A11,A12,A13,A21,A22,A23:double;
                A31,A32,A33,SIG:double; U1,U2,U3:ThreeD;
                returns ThreeD, ThreeD, ThreeD )
  for kx in 2,3
    V1,
    V2,
    V3 := for ky in 2,n
            DU1 := U1[kx,1,ky+1] - U1[kx,1,ky-1];
            DU2 := U2[kx,1,ky+1] - U2[kx,1,ky-1];
            DU3 := U3[kx,1,ky+1] - U3[kx,1,ky-1];

            V1  := U1[kx,1,ky] + A11 * DU1 + A12 * DU2 + A13 * DU3 +
                     SIG * (U1[kx+1,1,ky] - 2.0d0 * U1[kx,1,ky] + U1[kx-1,1,ky]);

            V2  := U2[kx,1,ky] + A21 * DU1 + A22 * DU2 + A23 * DU3 +
                     SIG * (U2[kx+1,1,ky] - 2.0d0 * U2[kx,1,ky] + U2[kx-1,1,ky]);

            V3  := U3[kx,1,ky] + A31 * DU1 + A32 * DU2 + A33 * DU3 +
                     SIG * (U3[kx+1,1,ky] - 2.0d0 * U3[kx,1,ky] + U3[kx-1,1,ky]);

          returns array of V1
                  array of V2
                  array of V3
          end for;
    M1 := array [1: V1 ];
    M2 := array [1: V2 ];
    M3 := array [1: V3 ];
  returns array of M1
          array of M2
          array of M3
  end for
end function


function Main( rep,n:integer; A11,A12,A13,A21,A22,A23:double;
               A31,A32,A33,SIG:double; U1in,U2in,U3in:ThreeD;
               returns ThreeD, ThreeD, ThreeD )
   for i in 1, rep
       U1, U2, U3 := Loop8( n, A11, A12, A13, A21, A22, A23,
                         A31, A32, A33, SIG, U1in, U2in, U3in );
   returns value of U1
           value of U2
           value of U3
   end for
end function
```

```
% LOOP 9
% Integrate Predictors
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

function Loop9( n:integer; CO,DM22,DM23,DM24,DM25:double;
               DM26,DM27,DM28:double; PX:TwoD returns OneD )
    for i in 1,n returns
    array of PX[3,i] +
             CO   * (PX[5,i]  +  PX[6,i])   +
             DM22 * PX[7,i]  + DM23 * PX[8,i]  +
             DM24 * PX[9,i]  + DM25 * PX[10,i] +
             DM26 * PX[11,i] + DM27 * PX[12,i] +
             DM28 * PX[13,i]
   end for
end function

function Main( rep,n:integer; CO,DM22,DM23,DM24,DM25:double;
               DM26,DM27,DM28:double; PXin:TwoD returns OneD )
   for i in 1,rep
     PXr := Loop9( n, CO, DM22, DM23, DM24, DM25, DM26, DM27, DM28, PXin )
   returns value of PXr
   end for
end function
```

```
% LOOP 10
% Difference Predictors
% Modified Parallel Algorithm
% SHOULD REWRITE FOR POINTER SWAP!!!

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

function Loop10( rep,n:integer; CX,PXin:TwoD returns TwoD )
let
  PX6, PX7, PX8, PX9, PX10, PX11, PX12, PX13, PX14 :=
  for i in 1, n
     PX5 := CX[5,i];
     PX6 := PX5 - PXin[5,i];
     PX7 := PX6 - PXin[6,i];
     PX8 := PX7 - PXin[7,i];
     PX9 := PX8 - PXin[8,i];
     PX10 := PX9 - PXin[9,i];
     PX11 := PX10 - PXin[10,i];
     PX12 := PX11 - PXin[11,i];
     PX13 := PX12 - PXin[12,i];
     PX14 := PX13 - PXin[13,i];
  returns array of PX6
          array of PX7
          array of PX8
          array of PX9
          array of PX10
          array of PX11
          array of PX12
          array of PX13
          array of PX14
  end for
in
  PXin[5:CX[5], PX6, PX7, PX8, PX9, PX10, PX11, PX12, PX13, PX14]
end let
end function

function Main( rep,n:integer; CX,PXin:TwoD returns TwoD )
let
   NewPX := for initial
               i := 1;
               PX := PXin;
           while ( i <= rep ) repeat
               i := old i + 1;
               PX := Loop10( i, n, CX, old PX );
           returns value of PX
           end for
in
```

```
  array_adjust( NewPX, 5, array_limh( NewPX ) )
end let
end function
```

```
% LOOP 11
% First Sum
% Sequential Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop11( n:integer; Yin:OneD returns OneD )
  for initial
      i := 2;
      X := Yin[1];
  while ( i <= n ) repeat
      i := old i + 1;
      X := old X + Yin[old i];
  returns array of X
  end for
end function

function Main( rep,n:integer; Yin:OneD returns OneD )
  for i in 1,rep
    Y := Loop11( n, Yin );
  returns value of Y
  end for
end function
```

```
% LOOP 12
% First Difference

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop12( n:integer; Y:OneD returns OneD )
  for i in 1,n returns
  array of Y[i+1] - Y[i]
  end for
end function

function Main( rep,n:integer; Yin:OneD returns OneD )
   for i in 1, rep
       Y := Loop12( n, Yin );
   returns value of Y
   end for
end function
```

```
% LOOP 13
% 2-D PIC  Particle In Cell

Define  Main

type double = double_real;
type IOneD  = array[integer];
type OneD   = array[double];
type TwoD   = array[OneD];

function MOD2N(i, j: integer  returns integer)
  if i < 0 then
     i - (i / j * j) + j / 2 + abs(j/2)
  else
     i - (i / j * j) + j / 2 - abs(j/2)
  end if
end function


function Loop13( n:integer;
                E,F:IOneD; B,C,Hin,Pin:TwoD;
                Y,Z:OneD returns TwoD,TwoD)
  for initial
      i := 0;
      H := Hin;
      P := Pin
  while i < n repeat
      i := old i + 1;
      i1 := 1 + MOD2N(Trunc(old P[1,i]),64);
      j1 := 1 + MOD2N(Trunc(old P[2,i]),64);
      P1 := old P[4,i: old P[4,i] + C[i1,j1];
                  3,i: old P[3,i] + B[i1,j1];
                  2,i: old P[2,i] + old P[4,i] + C[i1,j1];
                  1,i: old P[1,i] + old P[3,i] + B[i1,j1]];
      i2 := MOD2N(Trunc(P1[1,i]),64);
      j2 := MOD2N(Trunc(P1[2,i]),64);
      i3 := i2 + E[i2+32];
      j3 := j2 + F[j2+32];
      P := P1[1,i: P1[1,i] + Y[i2+32];
              2,i: P1[2,i] + Z[j2+32]];
      H := old H[i3,j3: old H[i3,j3] + 1.0d0]
  returns
      value of H
      value of P
  end for
end function

function Main(  rep,n:integer;
                E,F:IOneD; B,C,Hin,Pin:TwoD;
                Y,Z:OneD returns TwoD,TwoD)
   for initial
       i := 1;
       H := Hin;
       P := Pin;
   while ( i <= rep ) repeat
       i := old i + 1;
       H,P := Loop13( n, E, F, B, C, old H, old P, Y, Z )
   returns value of H
          value of P
   end for
end function
```

```
% LOOP 14
% 1-D PIC  Particle in Cell

Define  Main

type double = double_real;
type IOneD  = array[integer];
type OneD   = array[double];

function MOD2N(i, j: integer  returns integer)
  if i < 0 then
     i - (i / j * j) + j / 2 + abs(j/2)
  else
     i - (i / j * j) + j / 2 - abs(j/2)
  end if
end function

function Loop14( rep,n:integer; FLX:double;
                DEXin,EXin,GRD,RHIn : OneD;
                returns OneD,OneD,IOneD,IOneD,
                        OneD,OneD,OneD,OneD,OneD )
  let DEX1,EX1,IR1,IX1,RX1,VX1,XI1,XX1 :=
        for i in 1,n
            j := Trunc(GRD[i]);
            EX := EXin[j];
            DEX := DEXin[j];
            XI := Double_Real(j);
            VX := EX - DEX * XI;
            k  := Trunc(VX + FLX);
            IR := MOD2N(k,512) + 1;
            RX := VX + FLX - Double_Real(k);
            XX := VX + FLX - Double_Real(k) + Double_Real(IR)
        returns array of DEX
                array of EX
                array of IR
                array of j
                array of RX
                array of VX
                array of XI
                array of XX
        end for
  in  DEX1,EX1,IR1,IX1,RX1,VX1,XI1,XX1,
      for initial
          i := 0;
        RH := RHIn
      while i < n repeat
        i := old i + 1;
        RH := old RH[IR1[i]:
                     old RH[IR1[i]] - RX1[i] + 1.0d0,
                     old RH[IR1[i] + 1] + RX1[i]]
      returns value of RH
      end for
  end let
end function

function Main( rep,n:integer; FLX:double;
                DEX,EX,GRD, RHIn : OneD;
                returns OneD,OneD,IOneD,IOneD,
                        OneD,OneD,OneD,OneD,OneD )
   for initial
       i  := 1;
       v1 := array OneD  [];
       v3 := array IOneD [];
       v4 := v3; v2 := v1; v5 := v2; v6 := v2; v7 := v2; v8 := v2;
       RH := RHin;
   while ( i <= rep ) repeat
       i := old i + 1;
       v1,v2,v3,v4,v5,v6,v7,v8, RH :=
          Loop14( i, n, FLX, DEX, EX, GRD, old RH );
   returns value of v1
          value of v2
          value of v3
          value of v4
          value of v5
          value of v6
          value of v7
          value of v8
           value of RH
   end for
end function
```

```
% LOOP 15
% Casual Fortran. Development Version

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

% global fsqrt( x:double returns double )
global Sqrt( x:double returns double )

function Loop15( n:integer; VF,VG,VH:TwoD returns TwoD, TwoD )
let
  VS, VYc := for j in 2, 6
               VSrc,
               VYrc := for i in 2, n-1
                         VGj   := VG[j];
                         VGjm1 := VG[j-1];
                         VHj   := VH[j];
                         VHjp1 := VH[j+1];

                          S := if VF[j,i] >= VF[j-1,i] then
                                 let
                                   R := Max(VGj[i],VGj[i+1]);
                                   S := VF[j,i];
                                   T := 0.053d0;
                                 in
                                   sqrt(VHj[i] * VHj[i] + R*R) * T/S
                                 end let
                               else
                                 let
                                   R := Max(VGjm1[i],VGjm1[i+1]);
                                   S := VF[j-1,i];
                                   T := 0.073d0;
                                 in
                                   sqrt(VHj[i] * VHj[i] + R*R) * T/S
                                 end let
                               end if;


                          T := if VHjp1[i] > VHj[i] then
                                0.053d0
```

```
                              else
                               0.073d0
                              end if;

                         Y := if VF[j,i] >= VF[j,i-1] then
                                let
                                  R := Max(VHj[i],VHjp1[i]);
                                  S := VF[j,i];
                                in
                                  sqrt(VGj[i] * VGj[i] + R*R) * T/S
                                end let
                              else
                                let
                                  R := Max(VHj[i-1], VHjp1[i-1]);
                                  S := VF[j,i-1];
                                in
                                  sqrt(VGj[i] * VGj[i] + R*R) * T/S
                                end let
                              end if;
                       returns array of S
                               array of Y
                       end for;

              T := if VH[j+1,n] > VH[j,n] then
                     0.053d0
                    else
                     0.073d0
                    end if;

              LastY := if VF[j,n] >= VF[j,n-1] then
                          let
                            R := Max(VH[j,n],VH[j+1,n]);
                            S := VF[j,n];
                          in
                            sqrt(VG[j,n] * VG[j,n] + R*R) * T/S
                          end let
                        else
                          let
                            R := Max(VH[j,n-1], VH[j+1,n-1]);
                            S := VF[j,n-1];
                          in
                            sqrt(VG[j,n] * VG[j,n] + R*R) * T/S
                          end let
                        end if;

              VSr := array_addh( VSrc, 0.0d0 );
              VYr := array_addh( VYrc, LastY );
            returns array of VSr
                    array of VYr
            end for;
in
  VS, array_addh( VYc, array_fill( 2,n,0.0d0 ) )
end let
end function


function Main( rep,n:integer; VF,VG,VH:TwoD returns TwoD, TwoD )
   for i in 1, rep
      v1,v2 := Loop15( n, VF, VG, VH );
   returns value of v1
           value of v2
   end for
end function
```

```
% LOOP 16
% Monte Carlo Search Loop
% Parallel Algorithm

Define  Main

type double = double_real;
type IOneD  = array[integer];
type OneD   = array[double];
type TwoD   = array[OneD];

function Loop16( n:integer; R,S,T:double; D,PLAN:OneD;
                ZONE:IOneD returns integer,integer)
   % interchanged
   let Y := for j in 1,n cross i in 1,ZONE[1]
              j4 := 2 * (n * (i-1) + j - 1) + 3;
```

```
            j5 := ZONE[2 * (n * (i-1) + j - 1) + 3];
            C1 := if j5 < n/3 then
                     if PLAN[j5] < T then ZONE[j4-1]
                     elseif PLAN[j5] = T then 0
                     else -ZONE[j4-1]
                     end if
                  elseif j5 < 2*n/3 then
                     if PLAN[j5] < S then ZONE[j4-1]
                     elseif PLAN[j5] = S then 0
                     else -ZONE[j4-1]
                     end if
                  elseif j5 < n then
                     if PLAN[j5] < R then ZONE[j4-1]
                     elseif PLAN[j5] = R then 0
                     else -ZONE[j4-1]
                     end if
                  elseif j5 = n then 0
                  elseif let
                           test := D[j5] - (D[j5-1] * exp(T - D[j5-2], 2) +
                                   exp(S - D[j5-3], 2) + exp(R - D[j5-4], 2));
                        in
                          test < 0.0d0
                        end let
                        then ZONE[j4-1]
                  else -ZONE[j4-1]
                  end if
         returns value of least if C1 = 0 then j4
                                else 2 * n * ZONE[1] + 2
                                end if
         end for
   in  if Y = 2 * n * ZONE[1] + 2 then 1, 0
       else (Y - 3) / (2 * n) + 1, Y
       end if
   end let
end function

function Main( rep,n:integer; R,S,T:double; D,PLAN:OneD;
                ZONE:IOneD returns integer,integer)
   for initial
       i  := 1;
       v1 := 0;
       v2 := 0;
   while ( i <= rep ) repeat
       i := old i + 1;
       v1,v2 := Loop16( n, R, S, T, D, PLAN, ZONE );
   returns value of v1
           value of v2
   end for
end function
```

```
% LOOP 17
% Implicit, Conditional Computation

Define  Main

type double = double_real;
type IOneD  = array[integer];
type OneD   = array[double];

function Loop17( n:integer; VLIN,VLR,VSP,VSTP,VXNEin:OneD;
                returns OneD, OneD, OneD)
  for initial
      i := n;
      XNMt := 1.0d0 / 3.0d0;
      E6t := 1.03d0 / 3.07d0;

      E3 := XNMt * VLR[i] + VLIN[i];
      XNC := 5.0d0 / 3.0d0 * E3;
      XNEI := VXNEin[i];
      VXND := E6t;

      VE3, E6, VXNE, XNM :=
         if ( XNMt > XNC ) then
            let
               E6 := XNMt * VSP[i] + VSTP[i];
            in
               E6, E6, E6, E6
            end let
         elseif ( XNEI > XNC ) then
            let
```

```
              E6 := XNMt * VSP[i] + VSTP[i];
           in
              E6, E6, E6, E6
           end let
        else
           E3, E3 + E3 - XNMt, E3 + E3 - XNEI, E3 + E3 - XNMt
        end if;
  while i > 2 repeat
      i := old i - 1;

      E3 := old XNM * VLR[i] + VLIN[i];
      XNC := 5.0d0 / 3.0d0 * E3;
      XNEI := VXNEin[i];
      VXND := old E6;

      VE3, E6, VXNE, XNM :=
        if ( old XNM > XNC ) then
           let
              E6 := old XNM * VSP[i] + VSTP[i];
           in
              E6, E6, E6, E6
           end let
        elseif ( XNEI > XNC ) then
           let
              E6 := old XNM * VSP[i] + VSTP[i];
           in
              E6, E6, E6, E6
           end let
        else
           E3, E3 + E3 - old XNM, E3 + E3 - XNEI, E3 + E3 - old XNM
        end if;
  returns array of VXNE
          array of VE3
          array of VXND
  end for
end function

function Main( rep,n:integer; VLIN,VLR,VSP,VSTP,VXNEin:OneD;
                returns OneD, OneD, OneD)
   for i in 1, rep
       v1,v2,v3 := Loop17( n, VLIN, VLR, VSP, VSTP, VXNEin );
   returns value of v1
           value of v2
           value of v3
   end for
end function
```

```
% LOOP 18
% 2-D Explicit Hydrodynamic Fragment
% Sequential and Parallel Algorithm

Define  Main

type double = double_real;
type IOneD  = array[integer];
type OneD   = array[double];
type TwoD   = array[OneD];

function acopy( lo,hi:integer; V:OneD returns OneD )
  for i in lo,hi returns array of V[i] end for
end function

function Loop18( n:integer; S,T:double;
                 ZA,ZB,ZM,ZP,ZQ,ZR,ZU,ZV,ZZ:TwoD
                 returns TwoD,TwoD )
  let ZAcore, ZBcore :=
      for j in 2,6
        ZArc,ZBrc :=
           for i in 2,n
           returns array of
                     (ZP[j+1,i-1] + ZQ[j+1,i-1] - ZP[j,i-1] - ZQ[j,i-1]) *
                     (ZR[j,i] + ZR[j,i-1]) / (ZM[j,i-1] + ZM[j+1,i-1])
                  array of
                     (ZP[j,i-1] + ZQ[j,i-1] - ZP[j,i] - ZQ[j,i]) *
                     (ZR[j,i] + ZR[j-1,i]) / (ZM[j,i] + ZM[j,i-1])
            end for;
      returns array of array_addl( array_addh(ZArc, ZA[j,n+1]), ZA[j,1] )
              array of array_addl( array_addh(ZBrc, ZB[j,n+1]), ZB[j,1] )
      end for;
      ZA1 := acopy(1,7,ZA[1]);
      ZB1 := acopy(1,7,ZB[1]);
      ZA7 := acopy(1,7,ZA[7]);
      ZB7 := acopy(1,7,ZB[7]);

      ZANew := array_addl( array_addh( ZAcore, ZA7), ZA1 );
      ZBNew := array_addl( array_addh( ZBcore, ZB7), ZB1 );

      ZRNew, ZZNew :=
      for j in 2,6
        ZRr, ZZr :=
           for i in 2,n
             ZUNew := ZU[j,i] + S *
                      (ZANew[j,i]   * (ZZ[j,i] - ZZ[j,i+1]) -
                      ZANew[j,i-1] * (ZZ[j,i] - ZZ[j,i-1]) -
                      ZBNew[j,i]   * (ZZ[j,i] - ZZ[j-1,i]) +
                      ZBNew[j+1,i] * (ZZ[j,i] - ZZ[j+1,i]));
             ZVNew := ZV[j,i] + S *
                      (ZANew[j,i]   * (ZR[j,i] - ZR[j,i+1]) -
                      ZANew[j,i-1] * (ZR[j,i] - ZR[j,i-1]) -
                      ZBNew[j,i]   * (ZR[j,i] - ZR[j-1,i]) +
                      ZBNew[j+1,i] * (ZR[j,i] - ZR[j+1,i]))
           returns array of ZR[j,i] + T * ZUNew
                   array of ZZ[j,i] + T * ZVNew
           end for;
      returns array of ZRr
              array of ZZr
      end for;
  in
      ZRNew, ZZNew
  end let
end function

function Main( rep,n:integer; S,T:double;
                ZAin,ZBin,ZM,ZP,ZQ,ZRin,ZUin,ZVin,ZZin:TwoD
                returns TwoD,TwoD )
   for i in 1, rep
      ZR, ZZ :=
        Loop18( n, S, T, ZAin, ZBin, ZM, ZP, ZQ, ZRin, ZUin, ZVin, ZZin );
   returns value of ZR
           value of ZZ
   end for
end function
```

```
% LOOP 19
% General Linear Recurrence Equations
% Sequential Agorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop19( n:integer; STIn: double;
                SA, SB: OneD returns OneD, double  )
  let
      B5t, STB5t :=
        for initial
            k    := 1;
            B5   := SA[1] + STIn * SB[1];
            STB5 := B5 - STIn;
        while ( k < n ) repeat
            k    := old k + 1;
            B5   := SA[k] + old STB5 * SB[k];
            STB5 := B5 - old STB5;
        returns array of B5
                value of STB5
         end for
  in
     for initial
         i    := 1;
         B5   := B5t;
         STB5 := STB5t;
     while ( i <= n ) repeat
        k := n + 1 - old i;
         i := old i + 1;
         B5V := SA[k] + old STB5 * SB[k];
         B5  := old B5[k:B5V];
         STB5 := B5V - old STB5;
     returns value of B5
```

```
              value of STB5
        end for
    end let
end function

function Main( rep,n:integer; STB5in: double;
                SA, SB: OneD returns OneD, double  )
   for initial
       i    := 1;
       B5   := array OneD [];
       STB5 := STB5in;
   while ( i <= rep ) repeat
       i := old i + 1;
       B5,STB5 := Loop19( n, old STB5, SA, SB );
   returns value of B5
          value of STB5
   end for
end function
```

```
% LOOP 20
% Discrete Ordinates Transport: Conditional Recurrence on XX

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

function Loop20( n:integer; DK,S,T:double;
                XXin,G,U,V,VX,W,Y,Z:OneD returns OneD, OneD )
   for initial
       i := 1;
       DI := Y[1] - G[1] / (XXin[1] + DK);
       DN := if DI = 0.0d0 then 0.20d0
             else max(S, min(Z[1]/DI, T))
             end if;
       X  := (XXin[1] * (W[1] + DN * V[1]) + U[1]) / (VX[1] + DN * V[1]);
       XX := XXin[2: XXin[1] + DN * (X - XXin[1])];
   while i < n repeat
       i := old i + 1;
       DI := Y[i] - G[i] / (old XX[i] + DK);
       DN := if DI = 0.0d0 then 0.20d0
             else max(S, min(Z[i]/DI, T))
             end if;
       X  := (old XX[i] * (W[i] + DN * V[i]) + U[i]) / (VX[i] + DN * V[i]);
       XX := old XX[i+1: old XX[i] + DN * (X - old XX[i])];
   returns array of X
           value of XX
   end for
end function

function Main( rep,n:integer; DK,S,T:double;
                XXin,G,U,V,VX,W,Y,Z:OneD returns OneD, OneD )
   for initial
       i  := 1;
       X  := array OneD [];
       XX := XXin;
   while ( i <= rep ) repeat
       i := old i + 1;
       X, XX := Loop20( n, DK, S, T, old XX, G, U, V, VX, W, Y, Z );
   returns value of X
          value of XX
   end for
end function
```

```
% LOOP 21
% Matrix * Matrix Product
% Assumes transpose(VY) to allow vectorization

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

function Loop21( n:integer; CX,PX,VY:TwoD returns TwoD )
  for k in 1,25 cross j in 1,n returns
  array of PX[k,j] + for i in 1,25 returns
                     value of sum VY[i,k] * CX[k,j]
                     end for
  end for
end function

function Main( rep,n:integer; CX,PXin,VY:TwoD returns TwoD )
   for i in 1, rep
       PX := Loop21( n, CX, PXin, VY );
   returns value of PX
   end for
end function
```

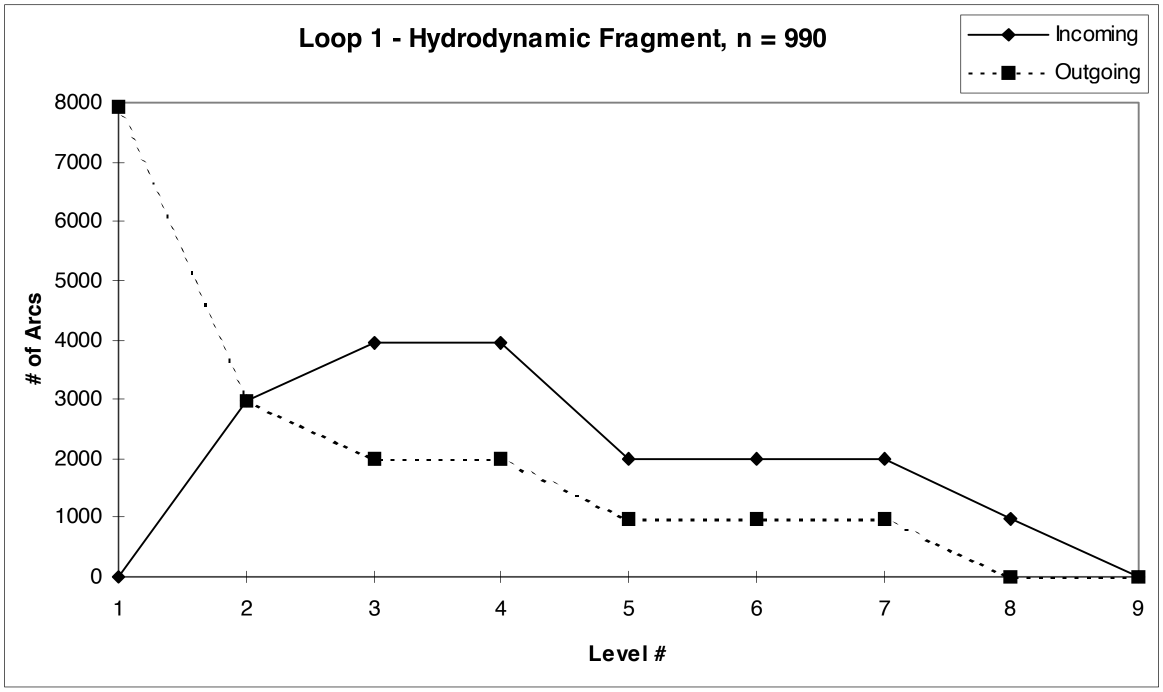
```
% LOOP 22
% Planckian Distribution

Define  Main

type double = double_real;
type OneD   = array[double];

global etothe( x:double returns double )
% global fexp( x:double returns double )

function Loop22( n:integer; U,V,X:OneD returns OneD, OneD )
  for k in 1,n
      Y := if ( U[k] < 20.0d0 * V[k] ) then
               U[k] / V[k]
           else
               20.0d0
           end if;
      W := X[k] / (etothe(Y) - 1.0d0);
  returns array of W
          array of Y
  end for
end function

function Main( rep,n:integer; U,V,X:OneD returns OneD, OneD )
   for i in 1, rep
       v1,v2 := Loop22( n, U,V,X );
   returns value of v1
           value of v2
   end for
end function
```

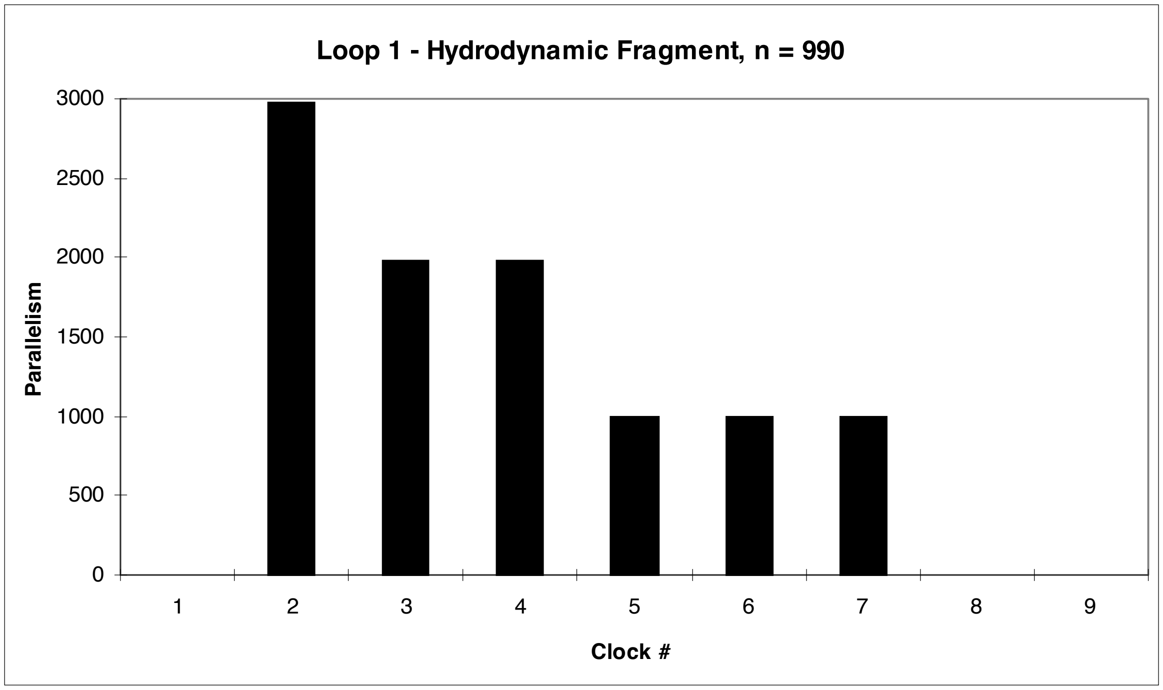
```
% LOOP 23
% 2-D Implicit Hydodynamics Fragment
% Sequential Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];
type TwoD   = array[OneD];

% transpose
function Loop23( n:integer; ZAin,ZB:TwoD;
                 ZR,ZU,ZV,ZZ:TwoD returns TwoD )
  for initial
      j   := 1;
      ZAt := ZAin;
  while ( j < 6 ) repeat
      j := old j + 1;

      ZArc := for initial
               k  := 1;
               ZA := old ZAt[j,1];
              while ( k < n ) repeat
               k := old k + 1;
               QA := old ZAt[j+1,k] * ZR[j,k] + old ZAt[j-1,k] * ZB[j,k] +
                     old ZAt[j,k+1] * ZU[j,k] + old ZA * ZV[j,k] +
                     ZZ[j,k];
               ZA := old ZAt[j,k] + 0.175d0 * (QA - old ZAt[j,k]);
              returns array of ZA
              end for;

      ZAt  := old ZAt[ j:array_addh( ZArc, old ZAt[j,n+1] ) ];
  returns value of ZAt
  end for
end function
```

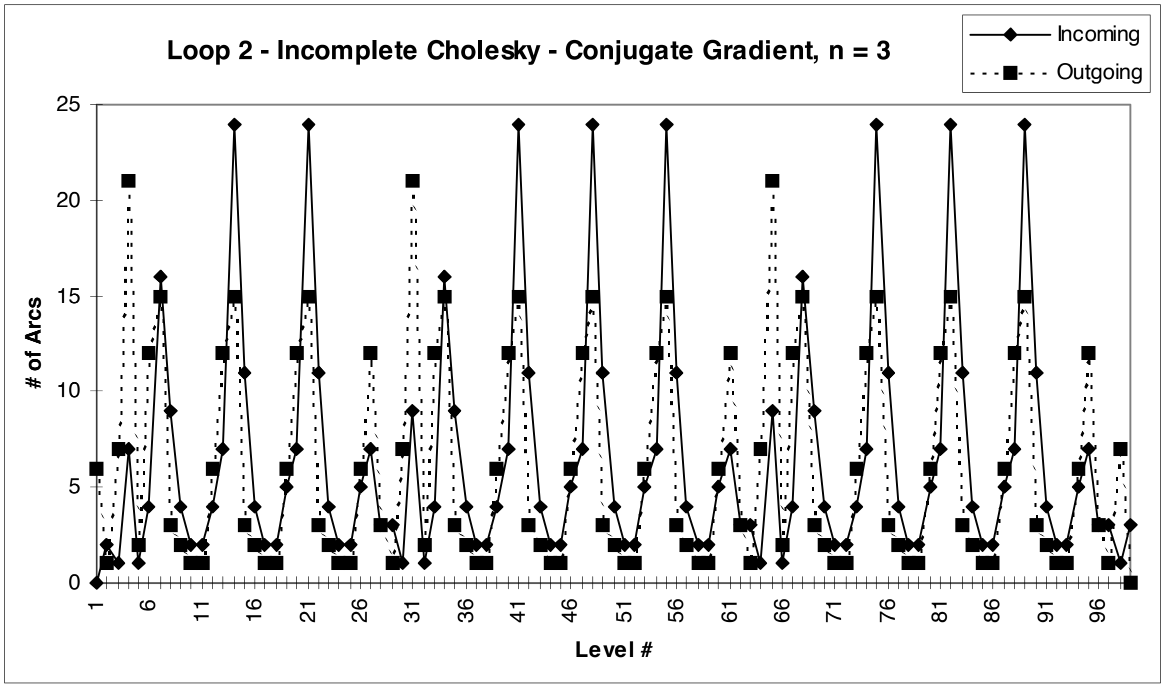
```
function Main(  rep,n:integer; ZAin,ZB:TwoD;
                ZR,ZU,ZV,ZZ:TwoD returns TwoD )
   for initial
       i := 1;
       ZA := ZAin;
   while ( i <= rep ) repeat
       i := old i + 1;
       ZA := Loop23( n, old ZA, ZB, ZR, ZU, ZV, ZZ );
   returns value of ZA
   end for
end function
```

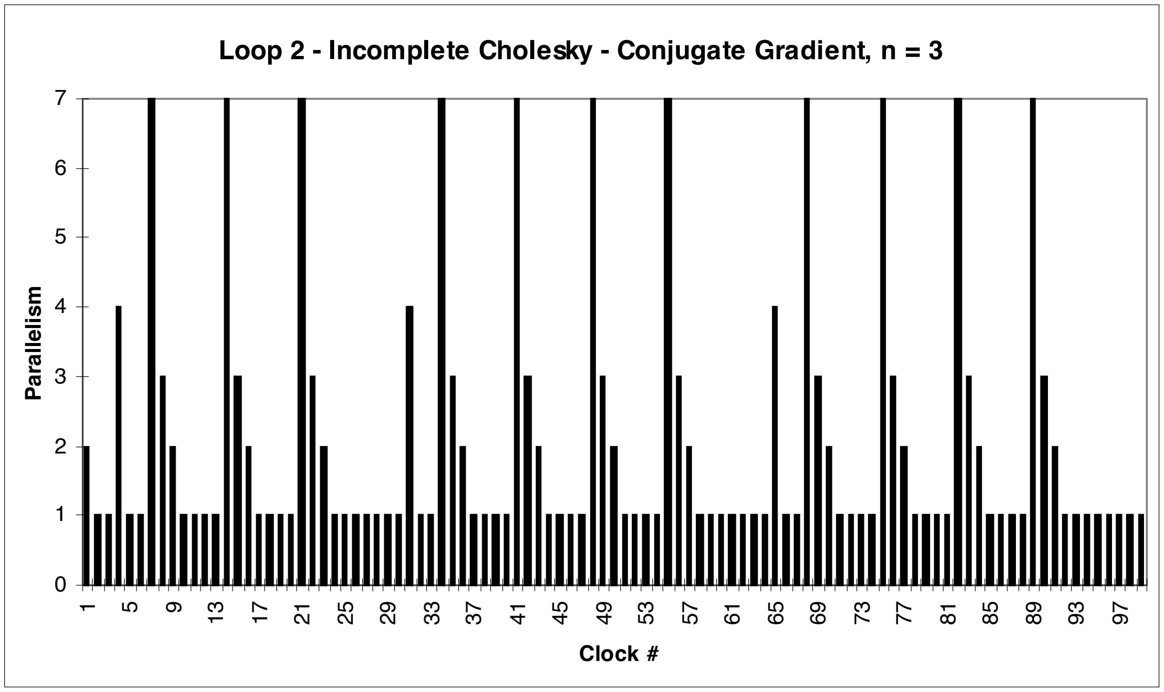
```
% LOOP 24
% Find Location of First Minimum in Array
% Vectorizable on Alliant
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop24( n:integer; X:OneD returns integer )
  for initial
    max24 := 1;
    k := 2;
  while ( k <= n ) repeat
    k := old k + 1;
    max24 := if ( X[old k] < X[old max24] ) then
               old k
             else
               old max24
             end if;
  returns value of max24
  end for
end function

function Main( rep,n:integer; X:OneD returns integer )
   for i in 1, rep
      v1 := Loop24( n, X );
   returns value of v1
   end for
end function
```

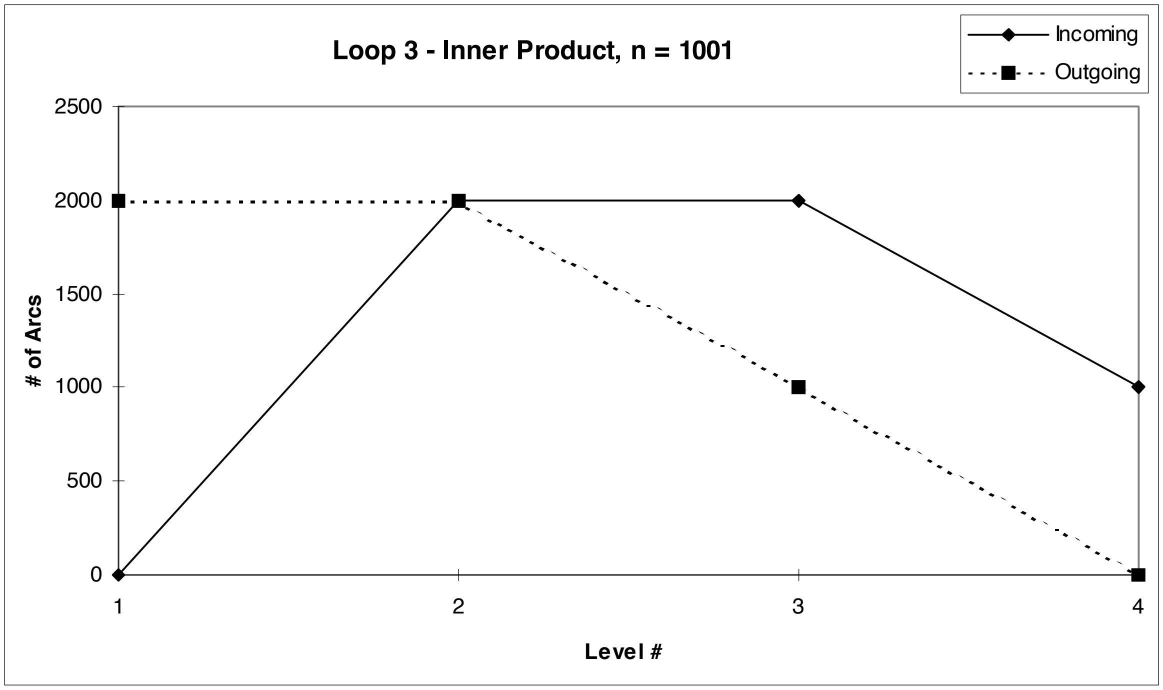
Appendix D
Simulation Result

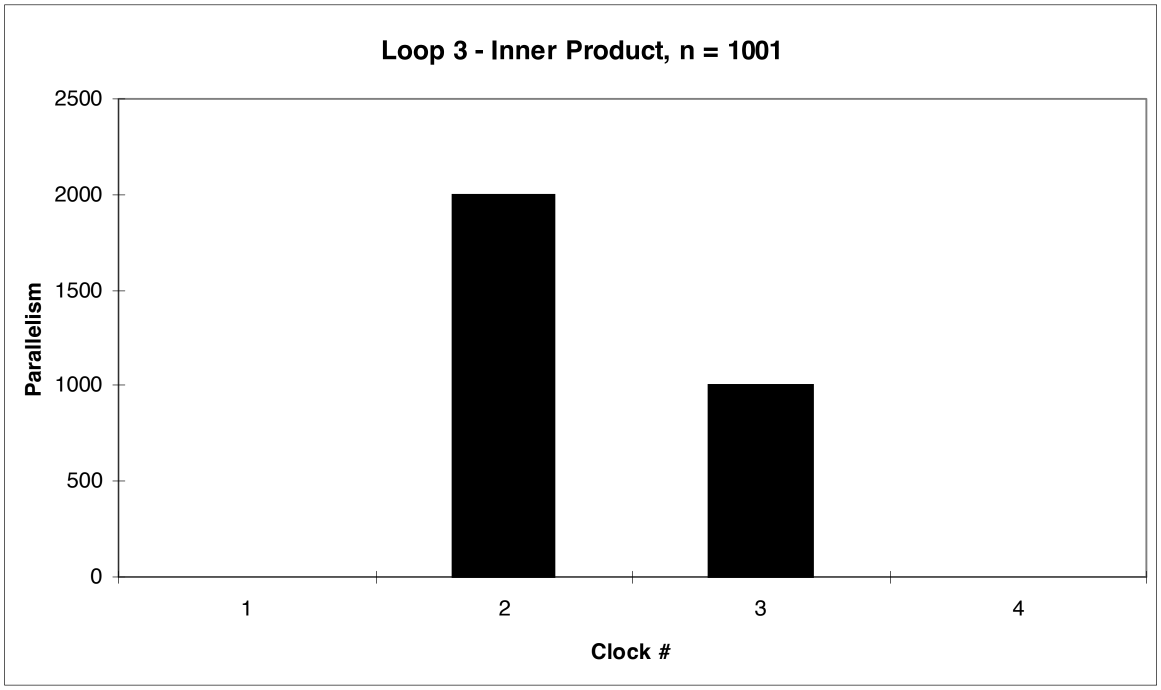


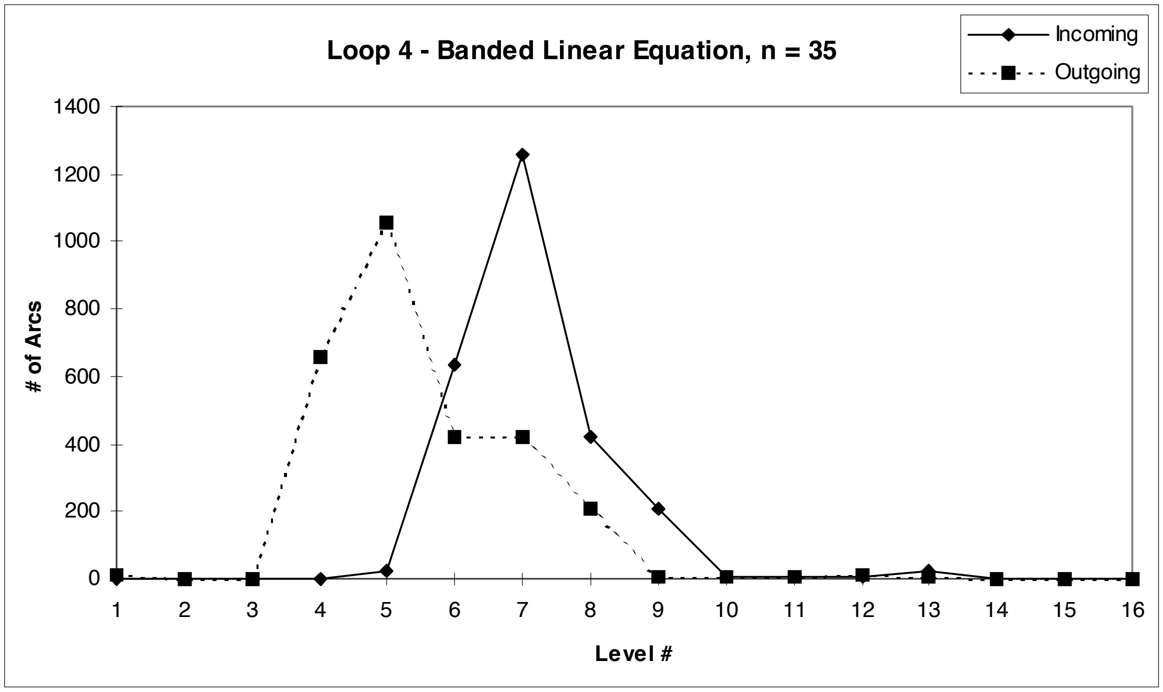



Loop 2 - Incomplete Cholesky - Conjugate Gradient, n = 3
Incoming
Outgoing
# of Arcs
Level #
0
5
10
15
20
25
1
6
11
16
21
26
31
36
41
46
51
56
61
66
71
76
81
86
91
96

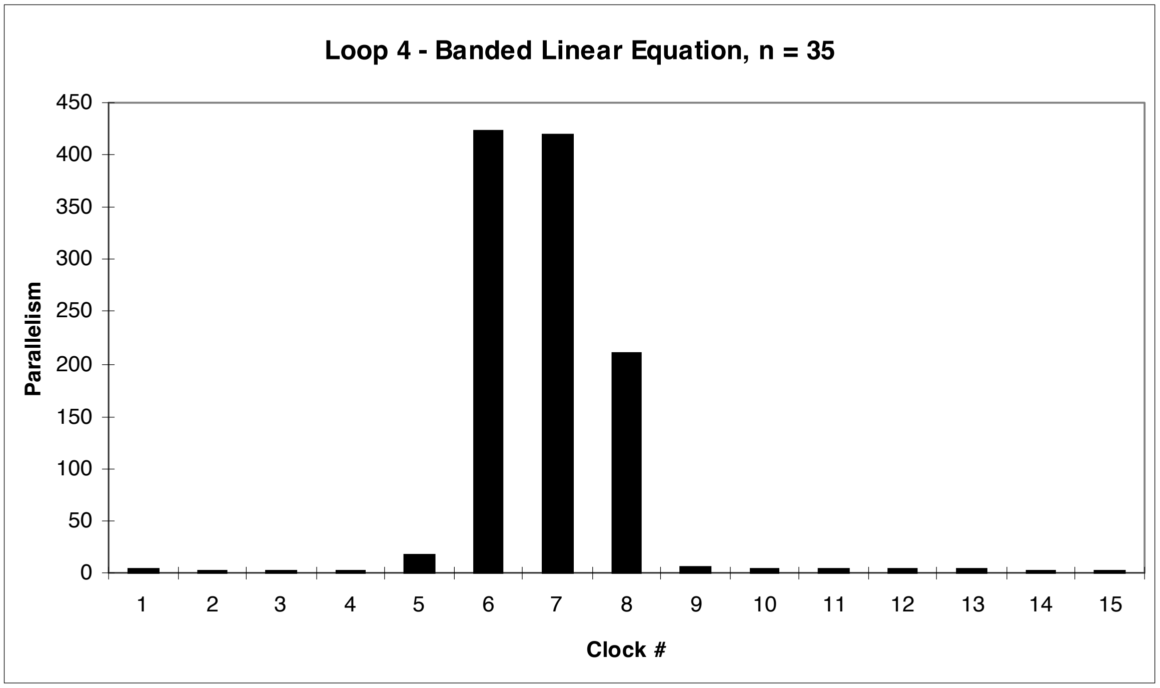

Loop 2 - Incomplete Cholesky - Conjugate Gradient, n = 3
Parallelism
Clock #
0
1
2
3
4
5
6
7
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97


Loop 3 - Inner Product, n = 1001
Incoming
Outgoing
# of Arcs
Level #
2500
2000
1500
1000
500
0
1
2
3
4

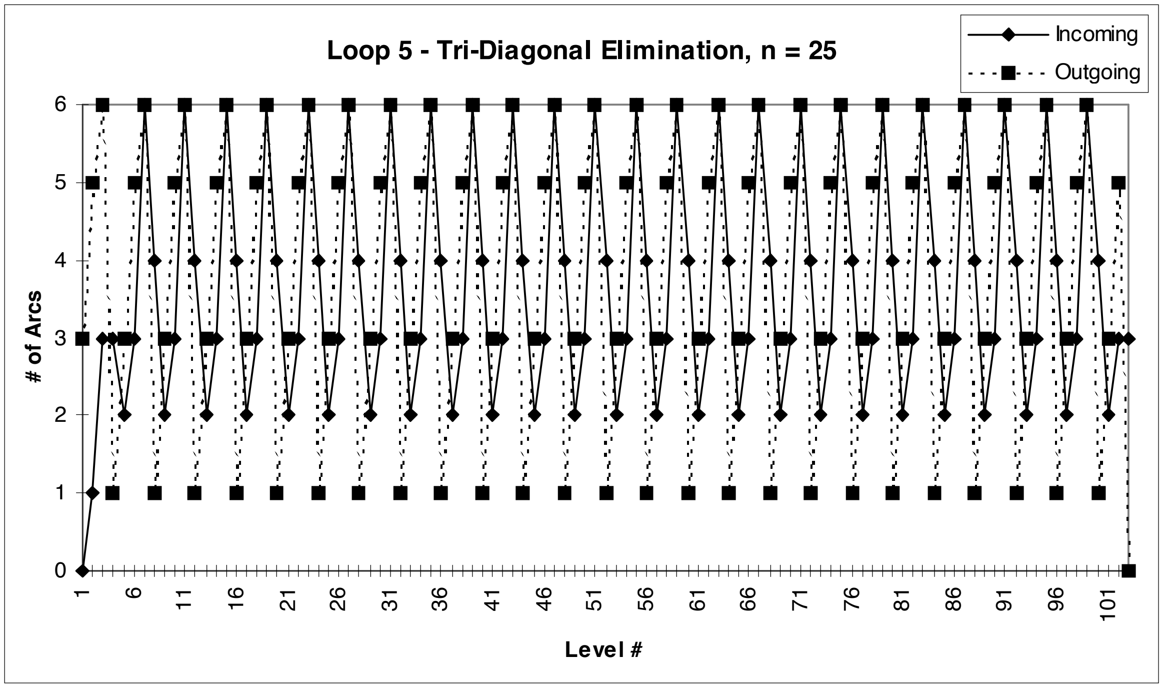

Loop 3 - Inner Product, n = 1001
Parallelism
Clock #
2500
2000
1500
1000
500
0
1
2
3
4



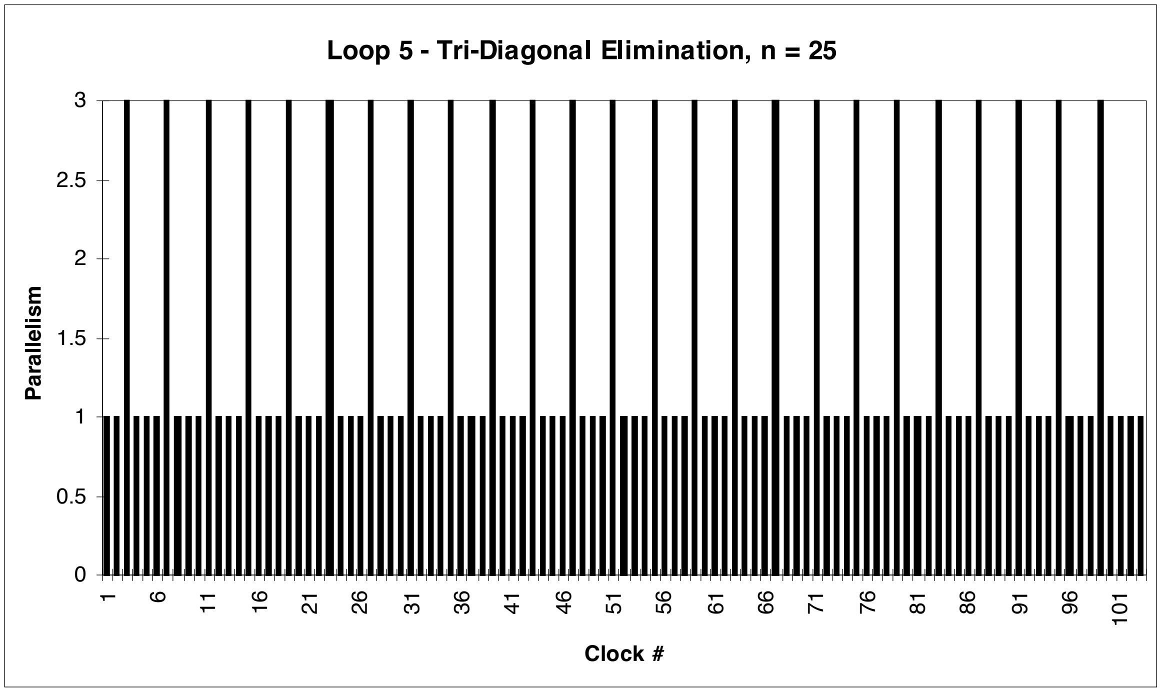
Loop 4 - Banded Linear Equation, n = 35
Incoming
Outgoing
# of Arcs
Level #
0
200
400
600
800
1000
1200
1400
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16



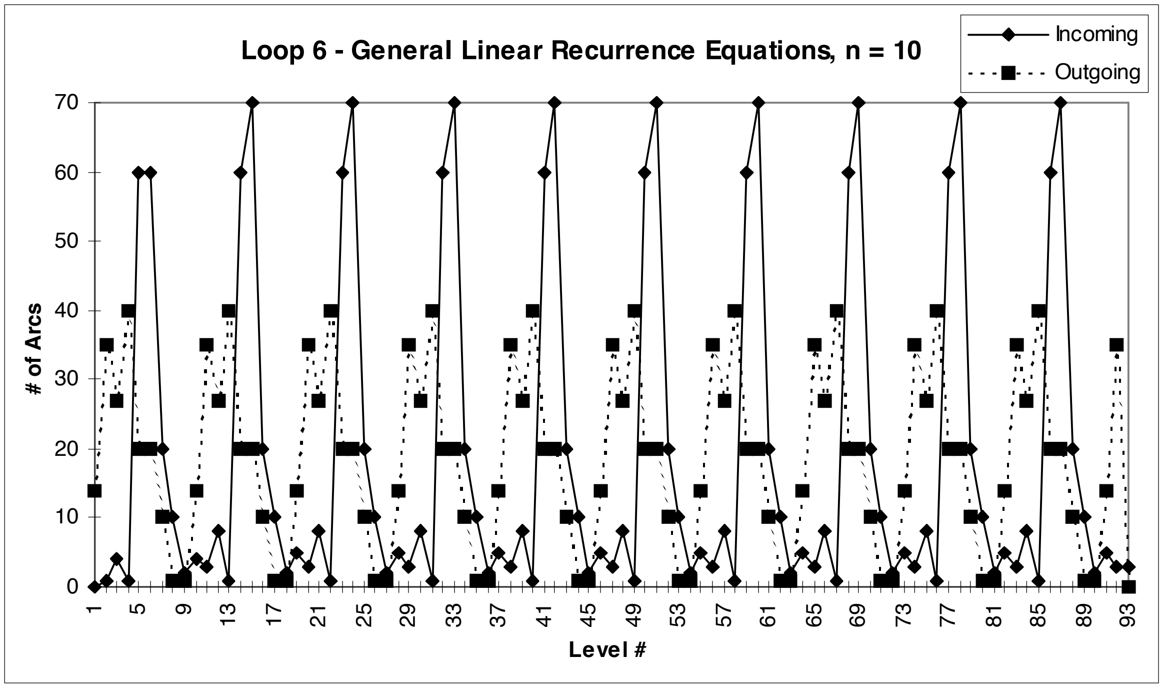
Loop 4 - Banded Linear Equation, n = 35
Parallelism
Clock #
0
50
100
150
200
250
300
350
400
450
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

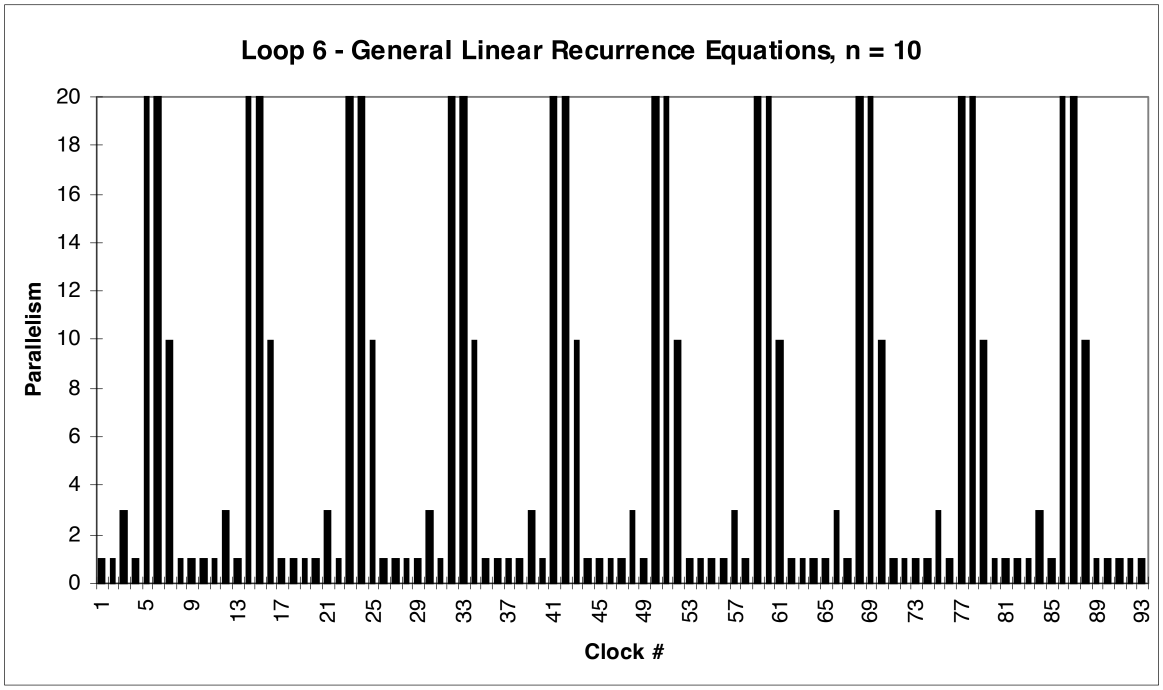


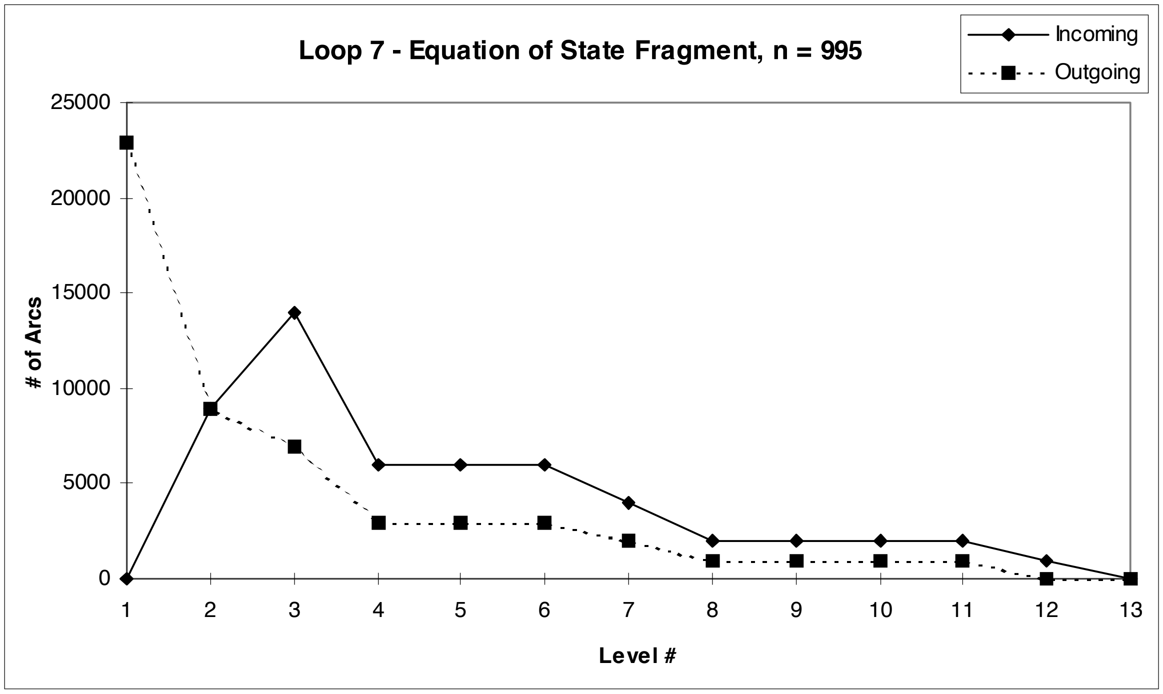
Loop 5 - Tri-Diagonal Elimination, n = 25
Incoming
Outgoing
# of Arcs
Level #

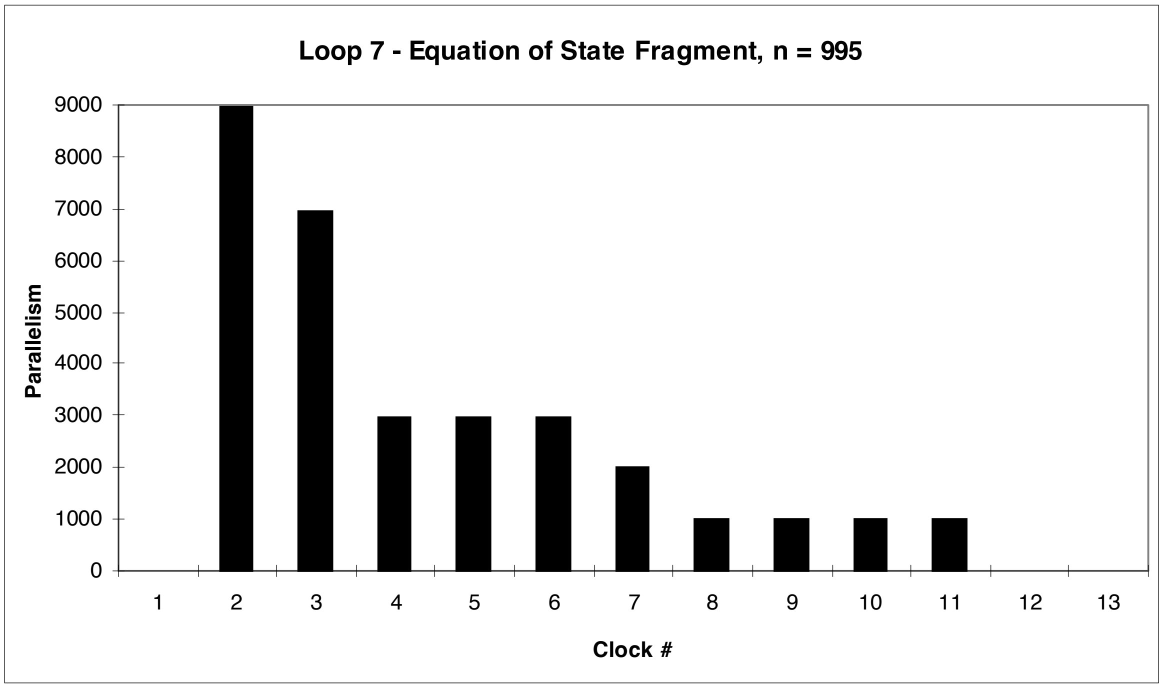


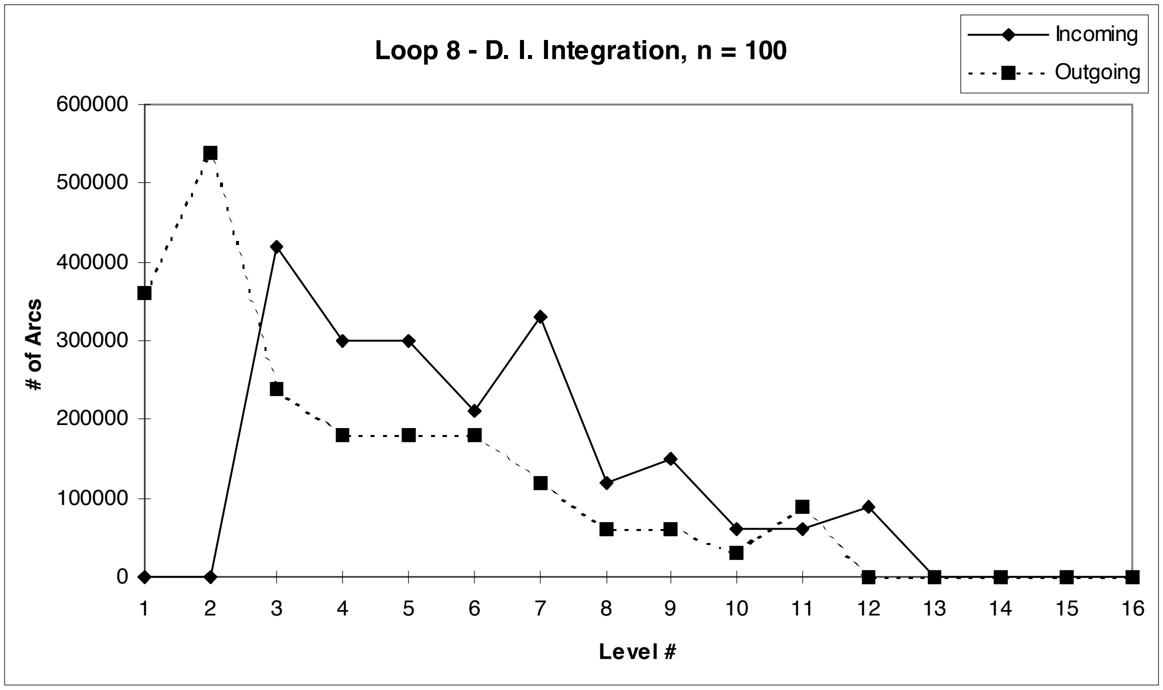
Loop 5 - Tri-Diagonal Elimination, n = 25
Parallelism
Clock #

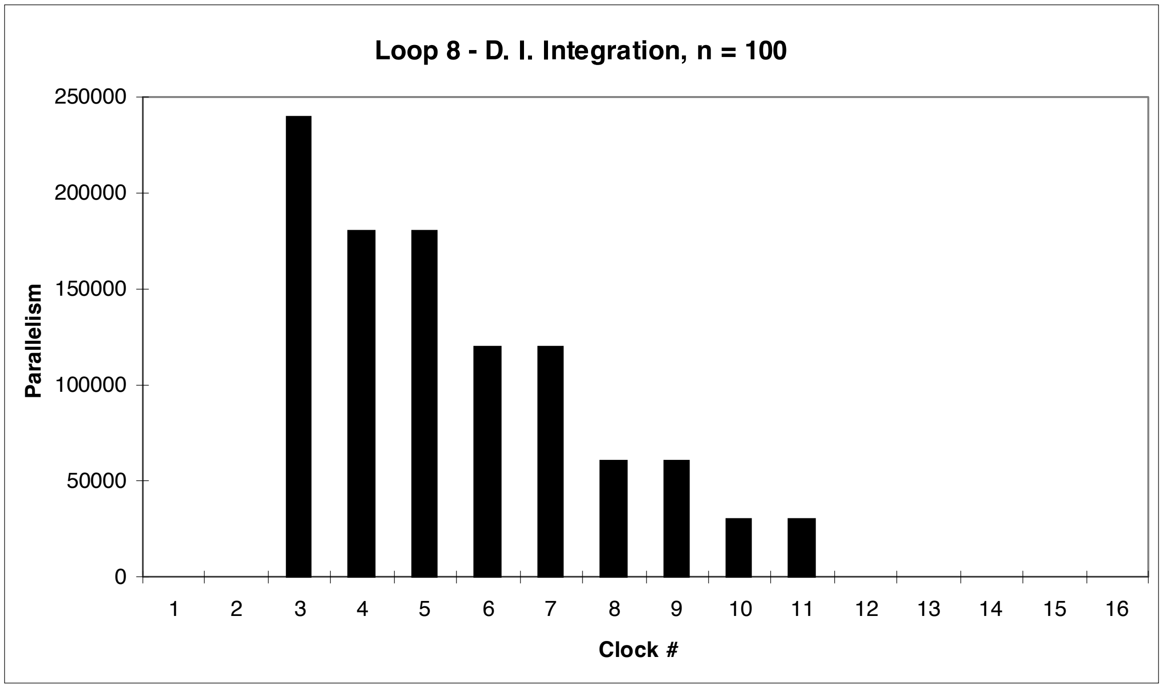

Loop 6 - General Linear Recurrence Equations, n = 10
Incoming
Outgoing
# of Arcs
Level #
0
10
20
30
40
50
60
70
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93

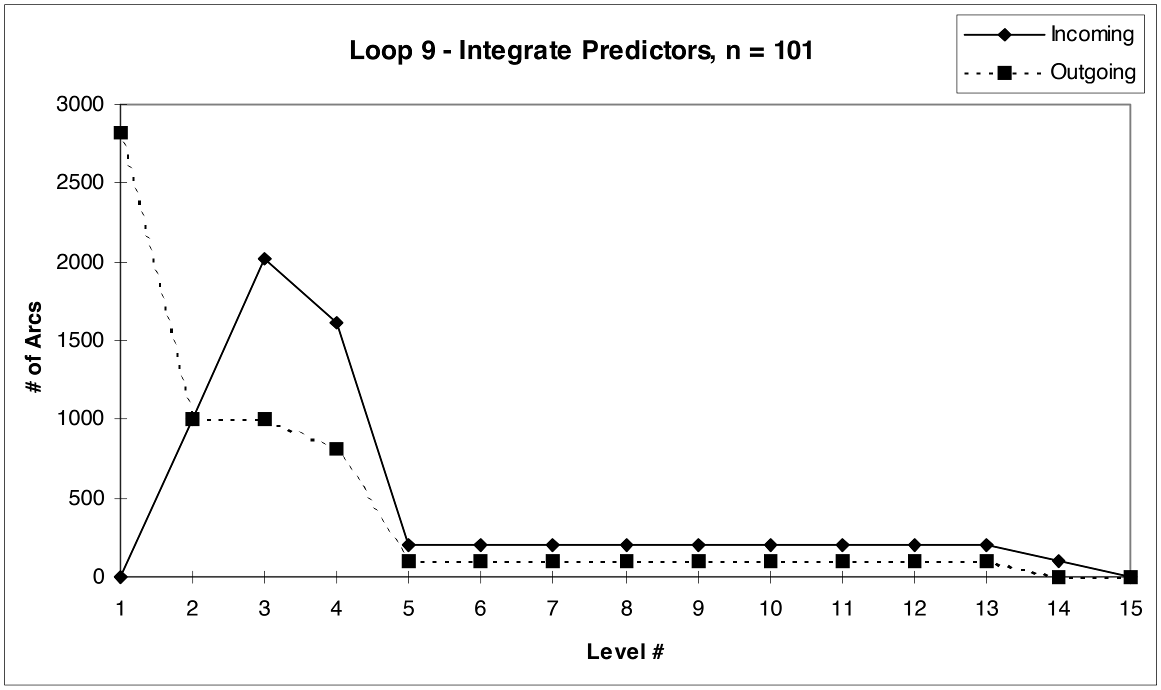

Loop 6 - General Linear Recurrence Equations, n = 10
Parallelism
Clock #
0
2
4
6
8
10
12
14
16
18
20
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93

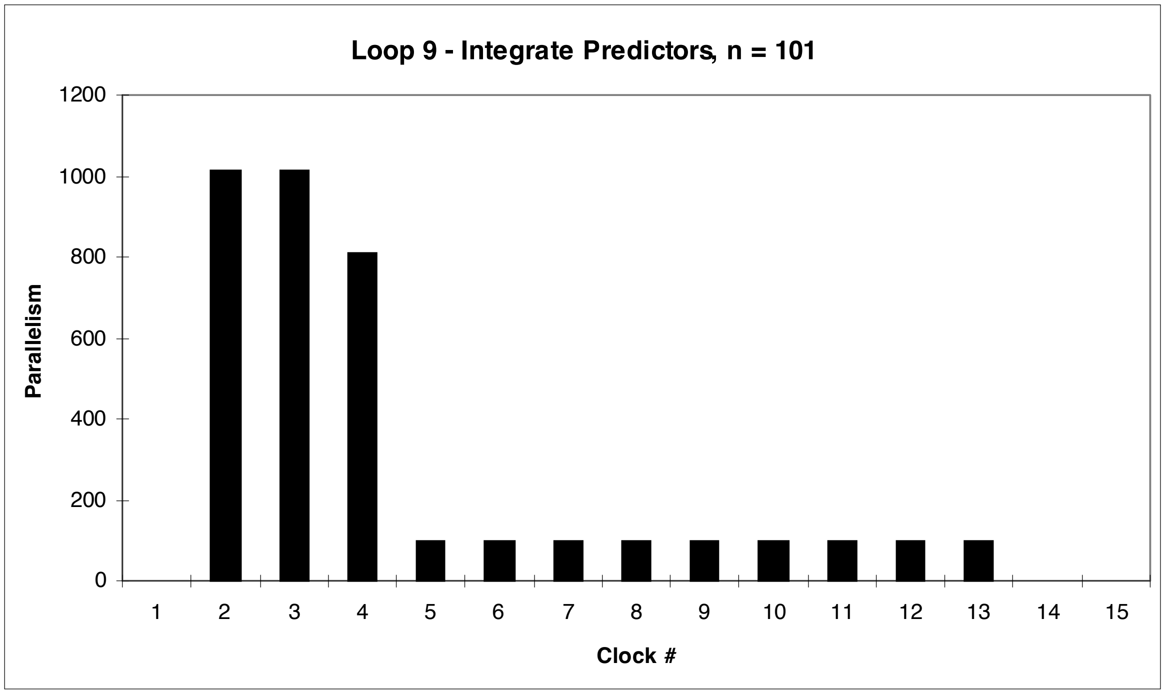

Loop 7 - Equation of State Fragment, n = 995
Incoming
Outgoing
# of Arcs
25000
20000
15000
10000
5000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Level #

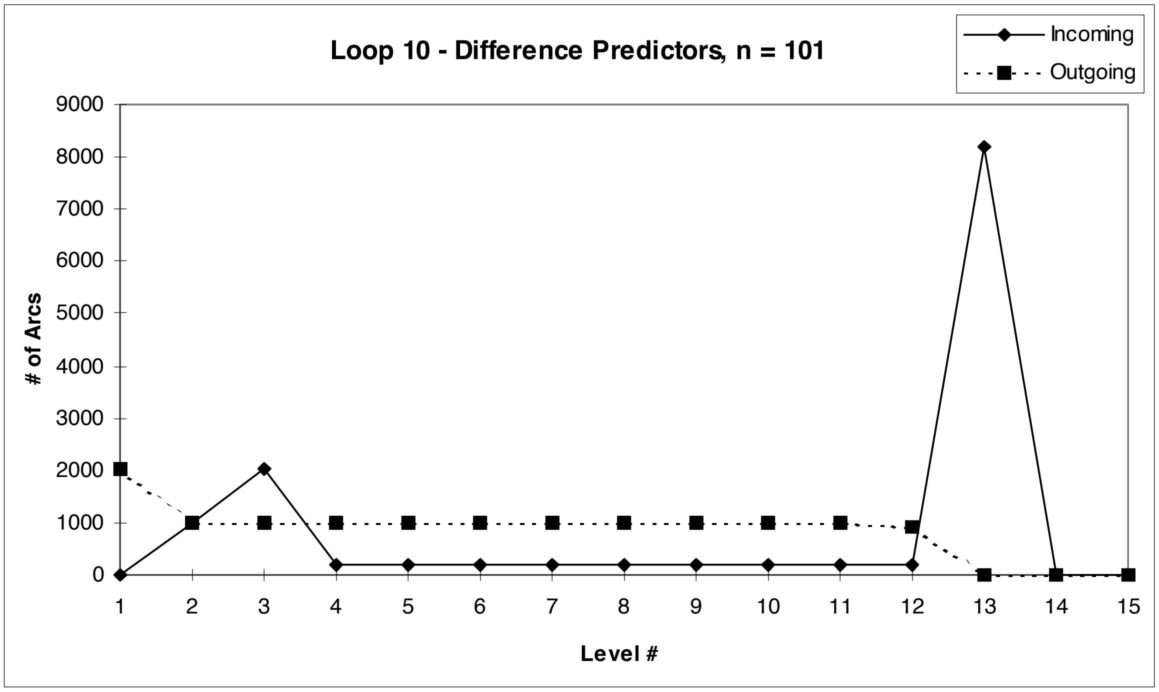

Loop 7 - Equation of State Fragment, n = 995
Parallelism
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Clock #


Loop 8 - D. I. Integration, n = 100
Incoming
Outgoing
# of Arcs
600000
500000
400000
300000
200000
100000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Level #

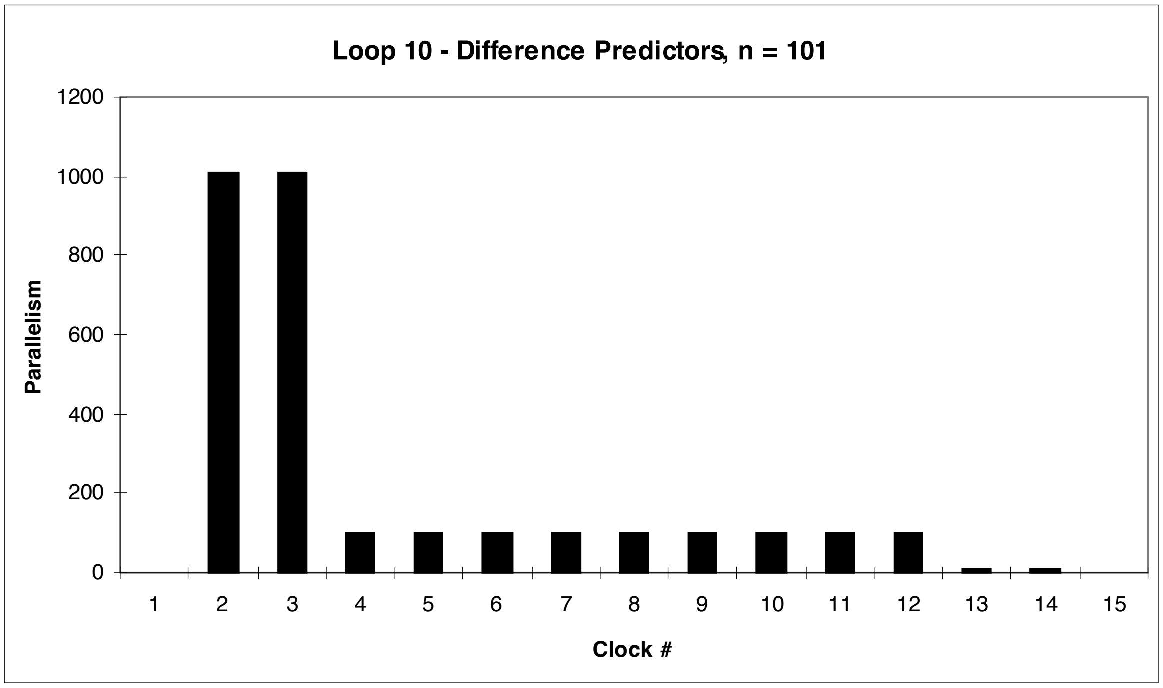

Loop 8 - D. I. Integration, n = 100
Parallelism
250000
200000
150000
100000
50000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Clock #


Loop 9 - Integrate Predictors, n = 101
Incoming
Outgoing
# of Arcs
Level #
3000
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

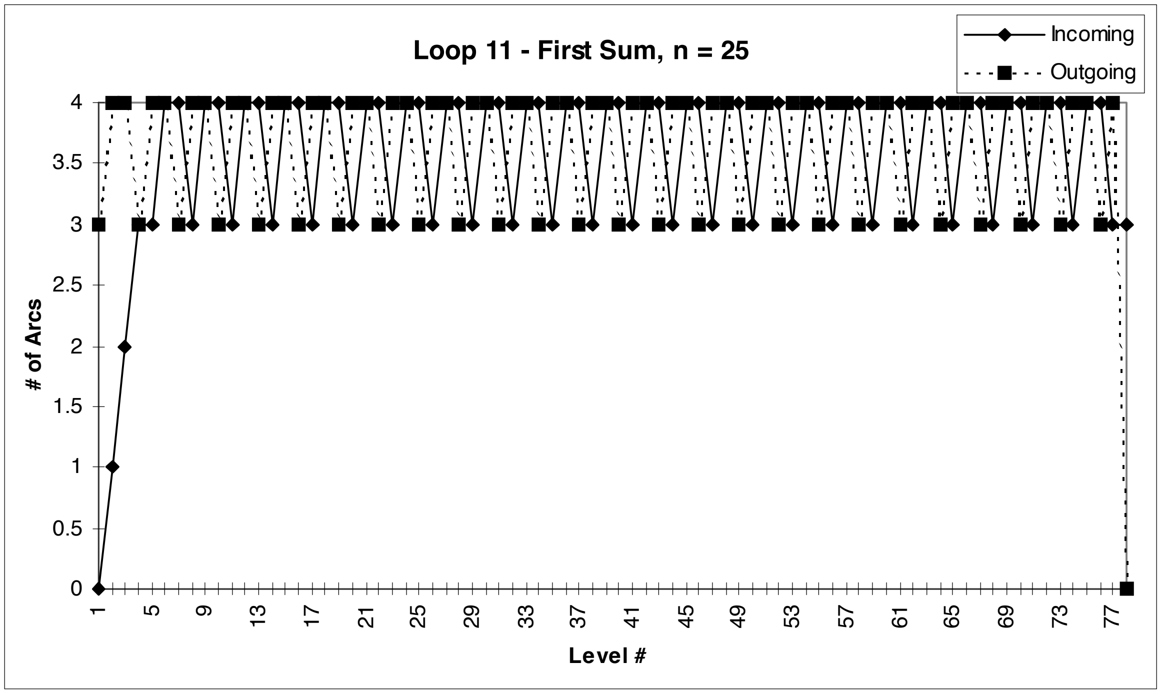

Loop 9 - Integrate Predictors, n = 101
Parallelism
Clock #
1200
1000
800
600
400
200
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

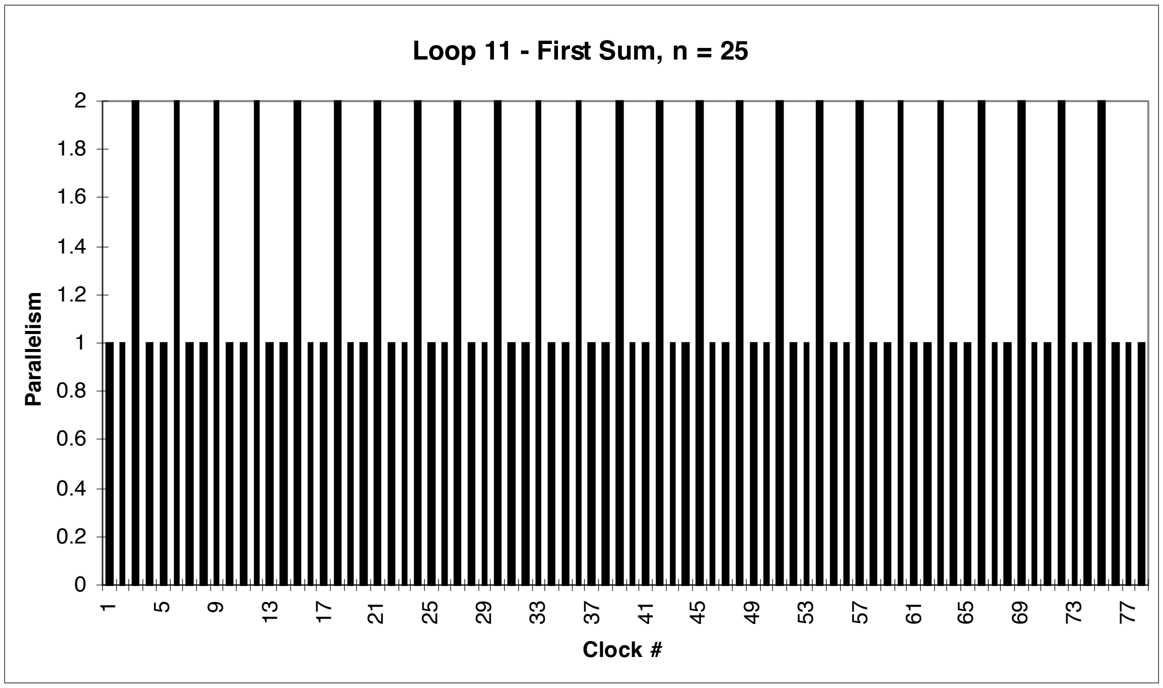


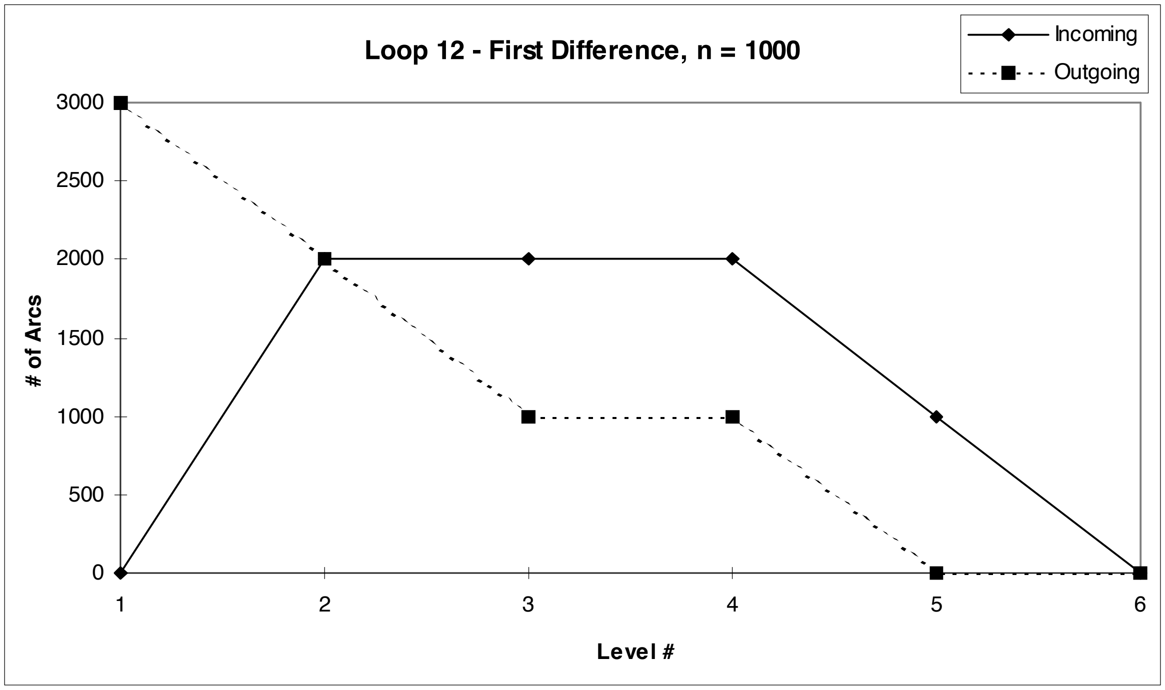
Loop 10 - Difference Predictors, n = 101
Incoming
Outgoing
# of Arcs
Level #
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

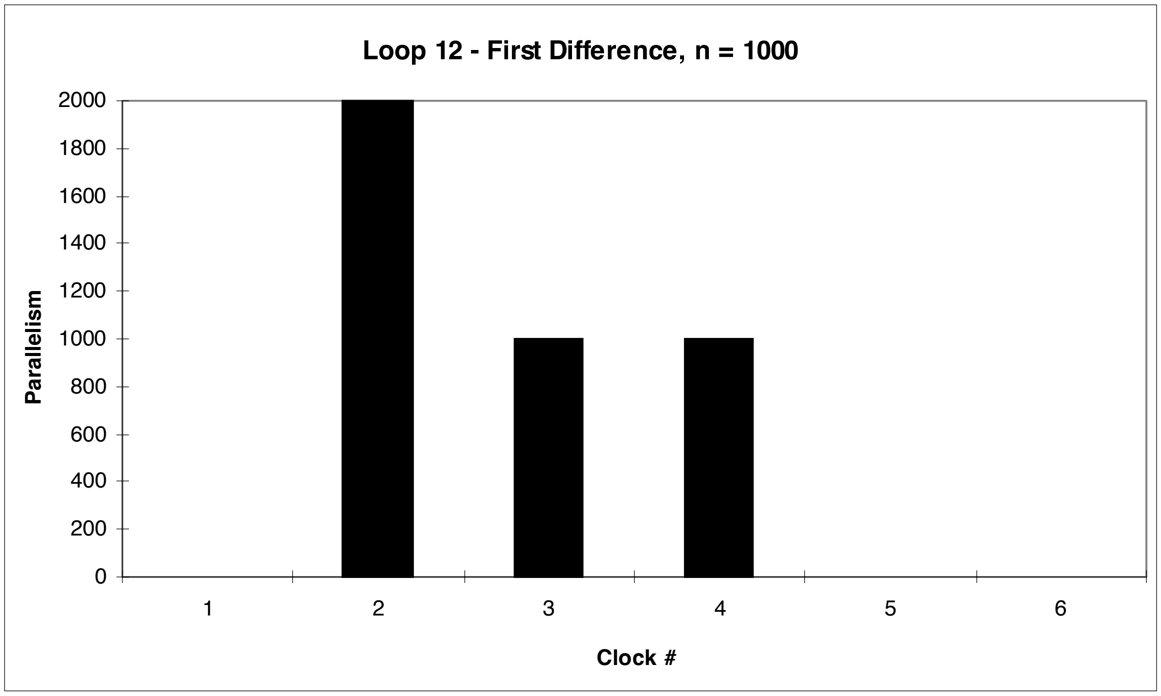


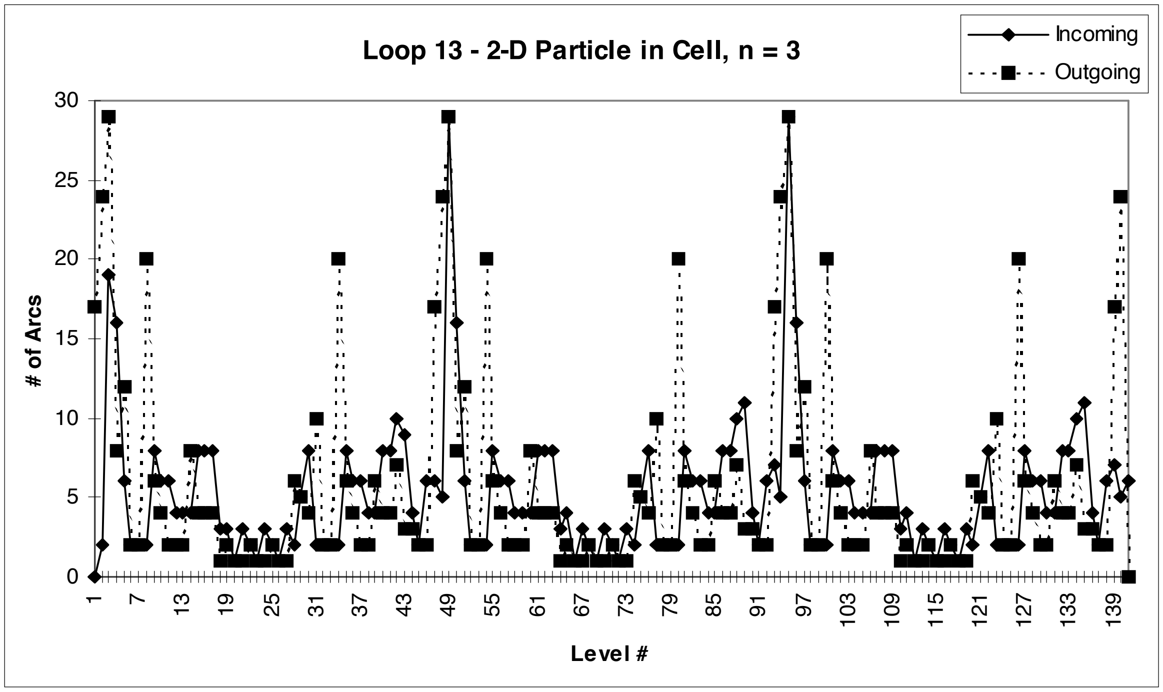
Loop 10 - Difference Predictors, n = 101
Parallelism
Clock #
0
200
400
600
800
1000
1200
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

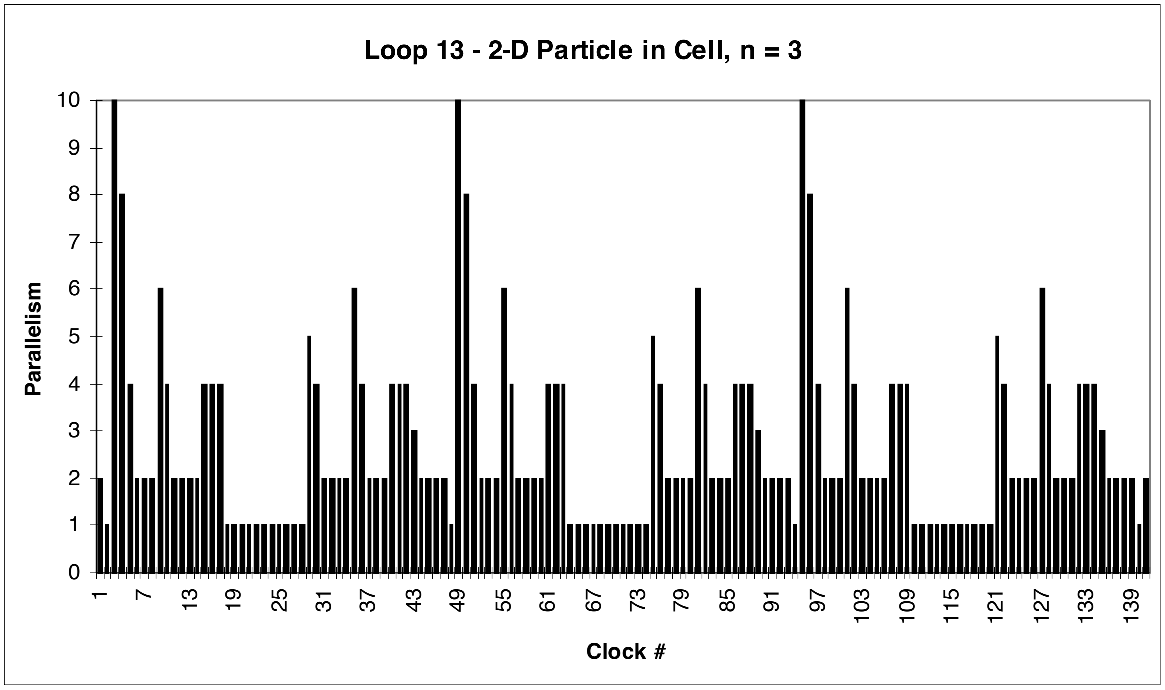

Loop 11 - First Sum, n = 25
Incoming
Outgoing
# of Arcs
Level #
0
0.5
1
1.5
2
2.5
3
3.5
4
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77

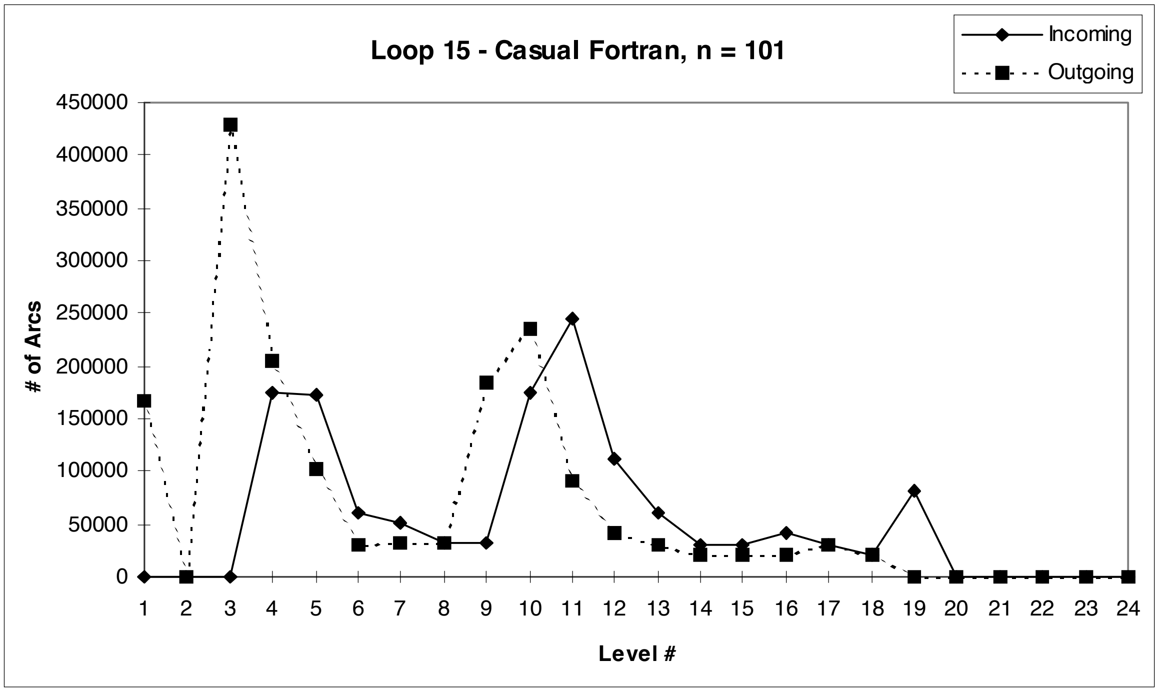

Loop 11 - First Sum, n = 25
Parallelism
Clock #
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77

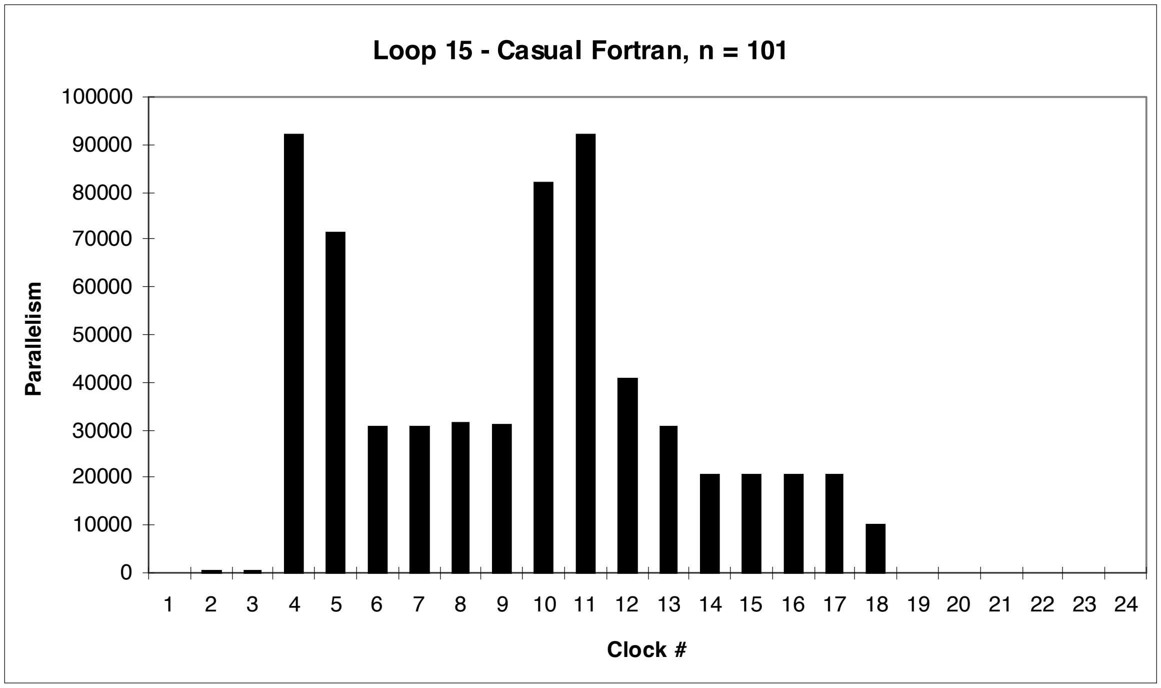

Loop 12 - First Difference, n = 1000
Incoming
Outgoing
3000
2500
2000
1500
1000
500
0
# of Arcs
1
2
3
4
5
6
Level #

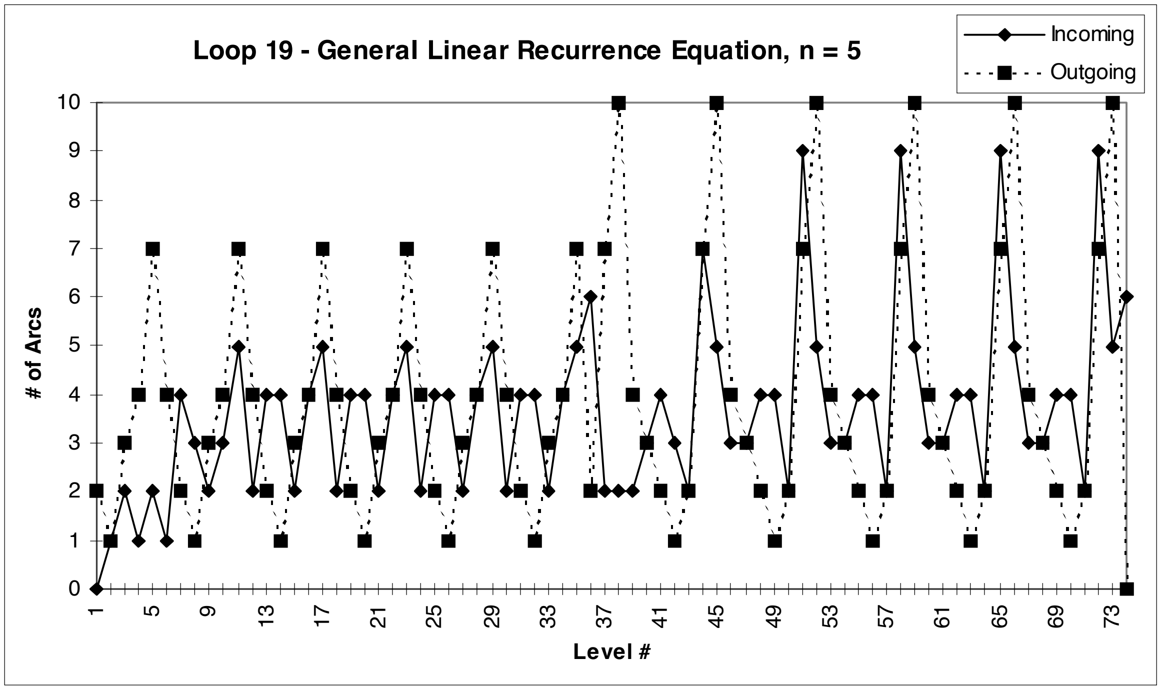

Loop 12 - First Difference, n = 1000
2000
1800
1600
1400
1200
1000
800
600
400
200
0
Parallelism
1
2
3
4
5
6
Clock #

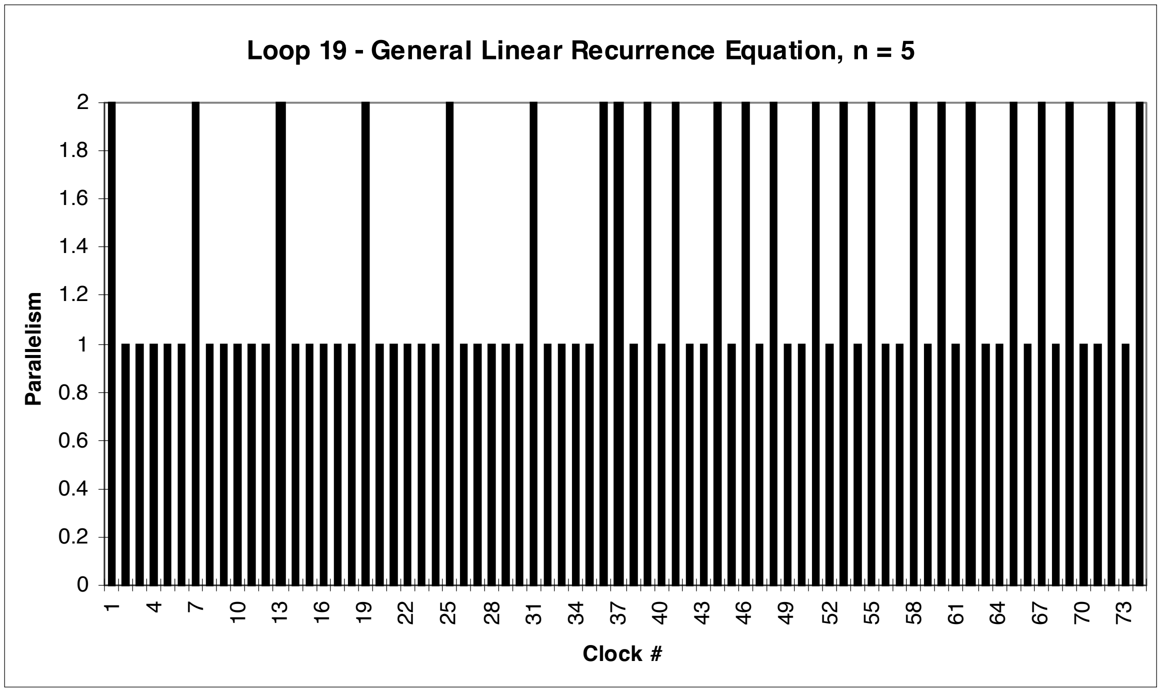

Loop 13 - 2-D Particle in Cell, n = 3
Incoming
Outgoing
# of Arcs
Level #
0
5
10
15
20
25
30
1
7
13
19
25
31
37
43
49
55
61
67
73
79
85
91
97
103
109
115
121
127
133
139

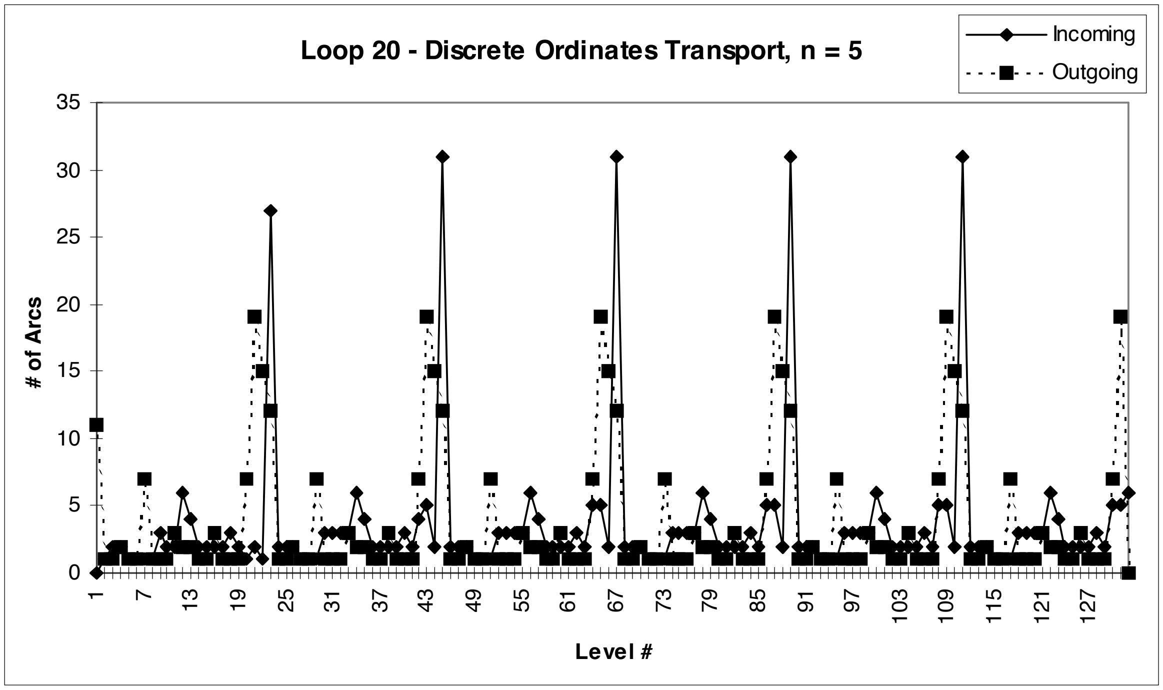

Loop 13 - 2-D Particle in Cell, n = 3
Parallelism
Clock #
0
1
2
3
4
5
6
7
8
9
10
1
7
13
19
25
31
37
43
49
55
61
67
73
79
85
91
97
103
109
115
121
127
133
139

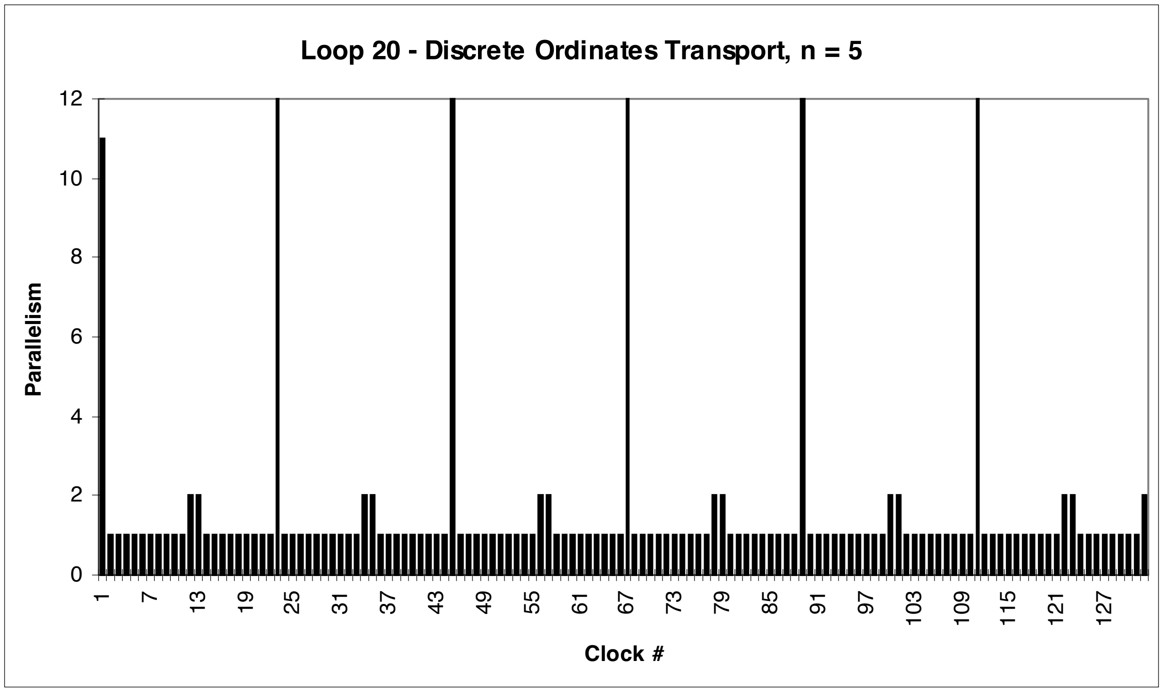

Loop 15 - Casual Fortran, n = 101
Incoming
Outgoing
# of Arcs
Level #
0
50000
100000
150000
200000
250000
300000
350000
400000
450000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

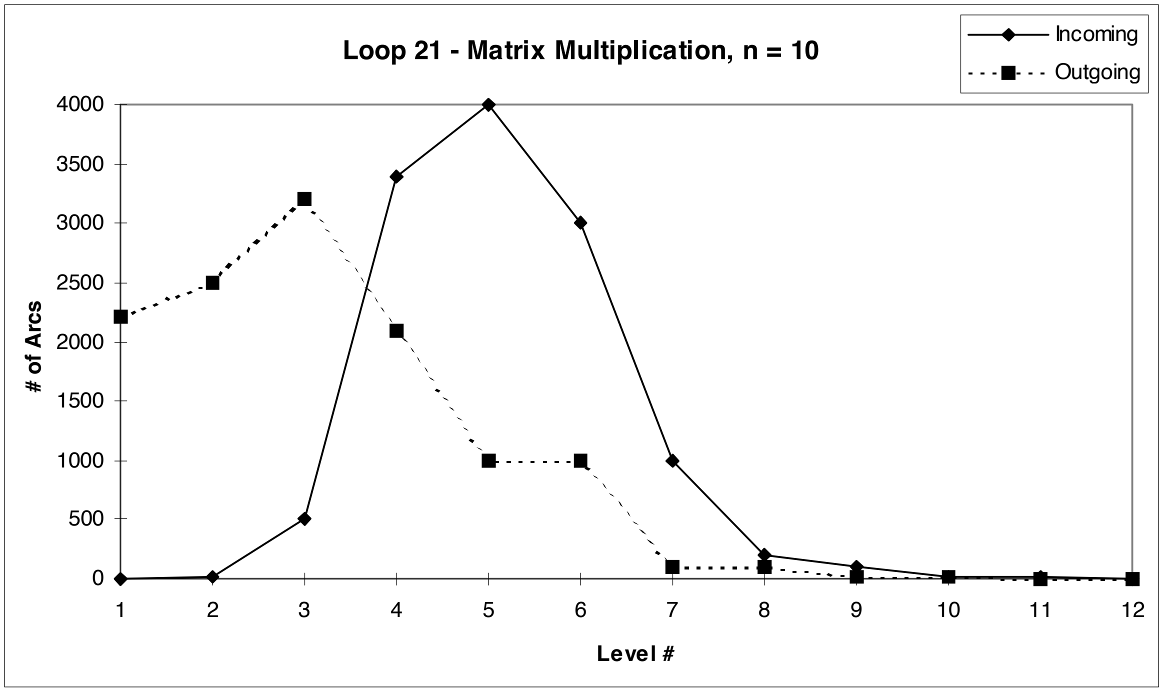

Loop 15 - Casual Fortran, n = 101
Parallelism
Clock #
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

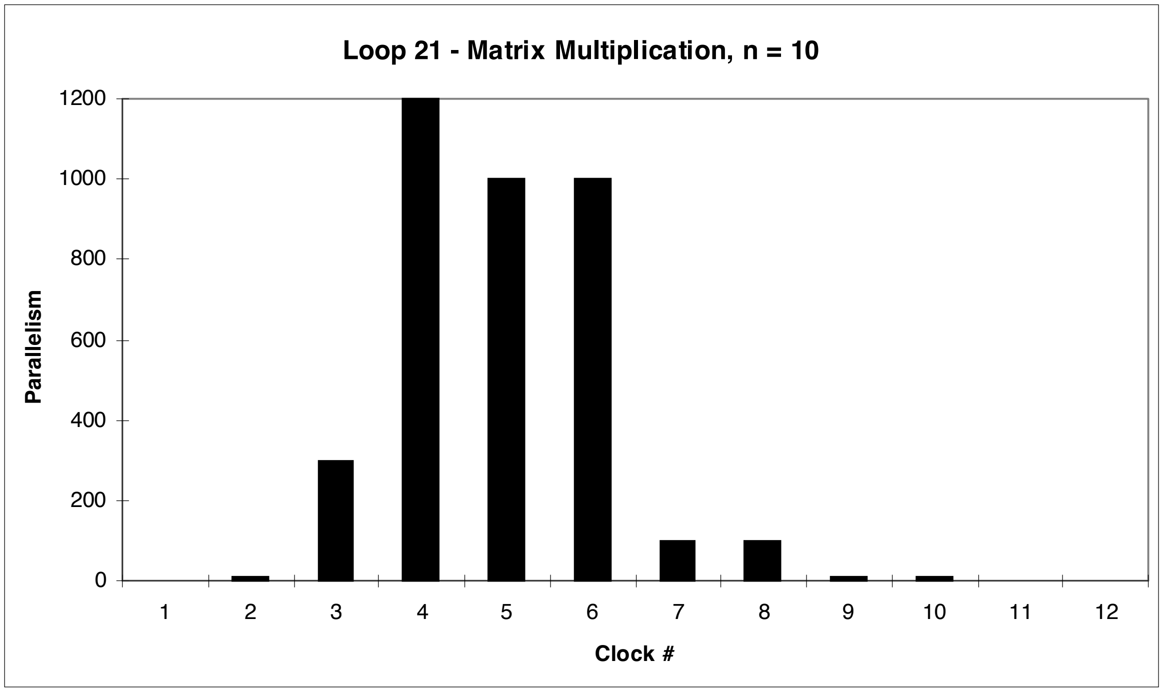

Loop 19 - General Linear Recurrence Equation, n = 5
Incoming
Outgoing
# of Arcs
Level #
0
1
2
3
4
5
6
7
8
9
10
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73

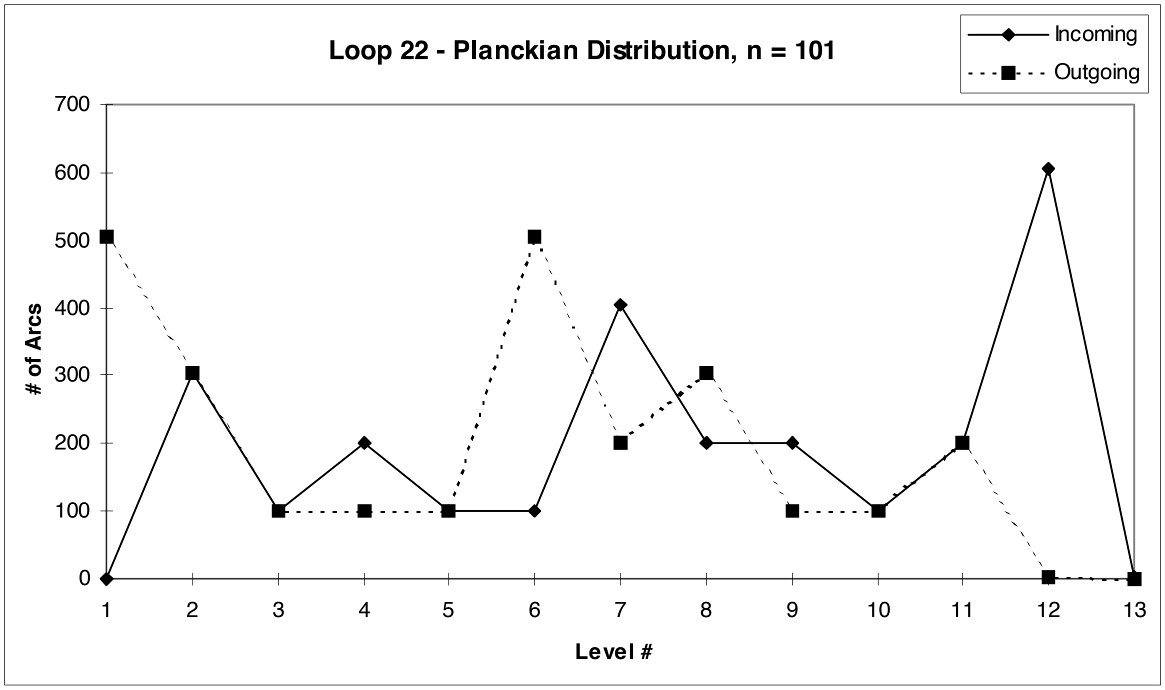

Loop 19 - General Linear Recurrence Equation, n = 5
Parallelism
Clock #
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73

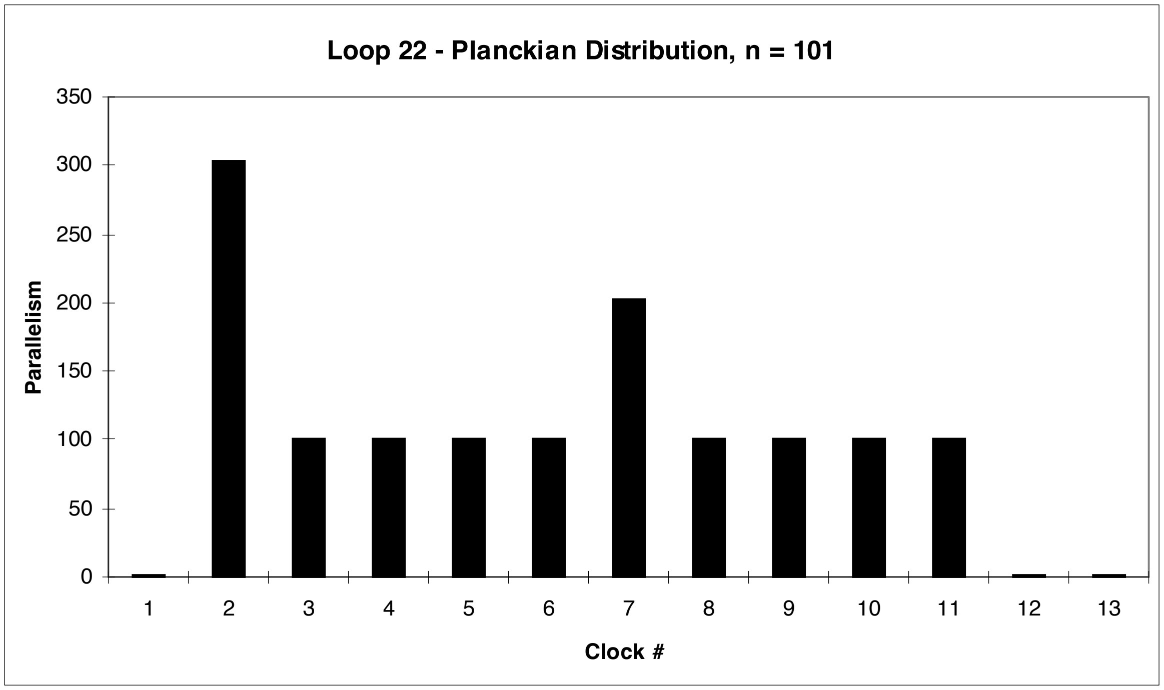

Loop 20 - Discrete Ordinates Transport, n = 5
Incoming
Outgoing
# of Arcs
Level #
0
5
10
15
20
25
30
35
1
7
13
19
25
31
37
43
49
55
61
67
73
79
85
91
97
103
109
115
121
127

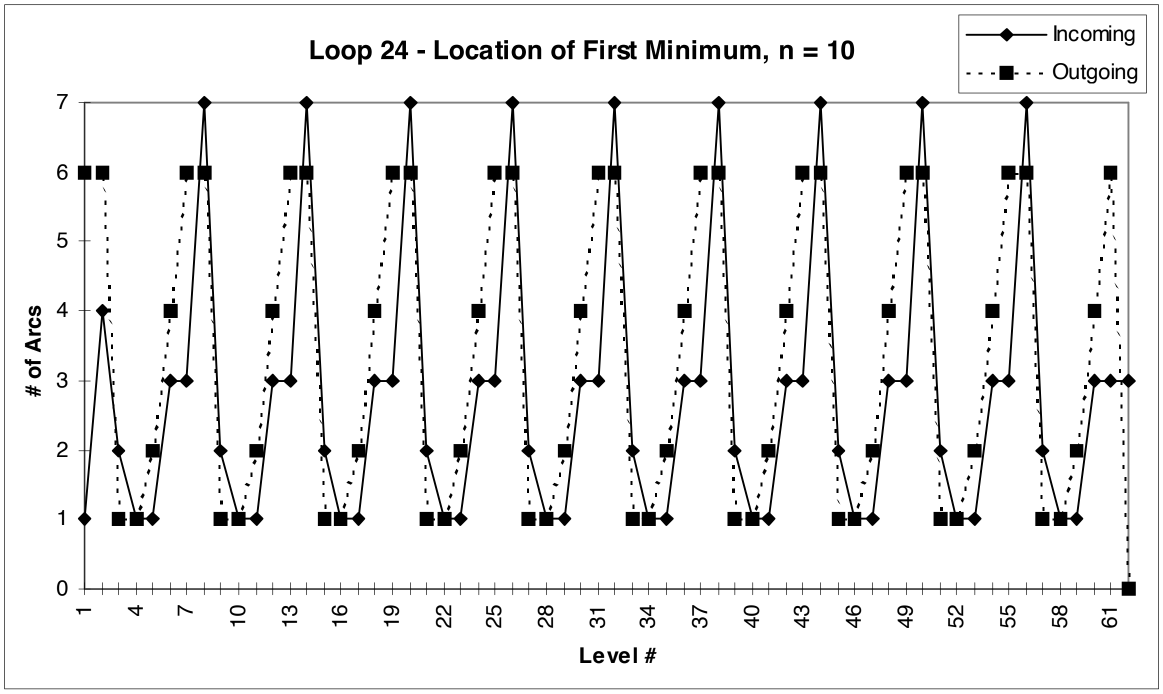

Loop 20 - Discrete Ordinates Transport, n = 5
Parallelism
Clock #
0
2
4
6
8
10
12
1
7
13
19
25
31
37
43
49
55
61
67
73
79
85
91
97
103
109
115
121
127

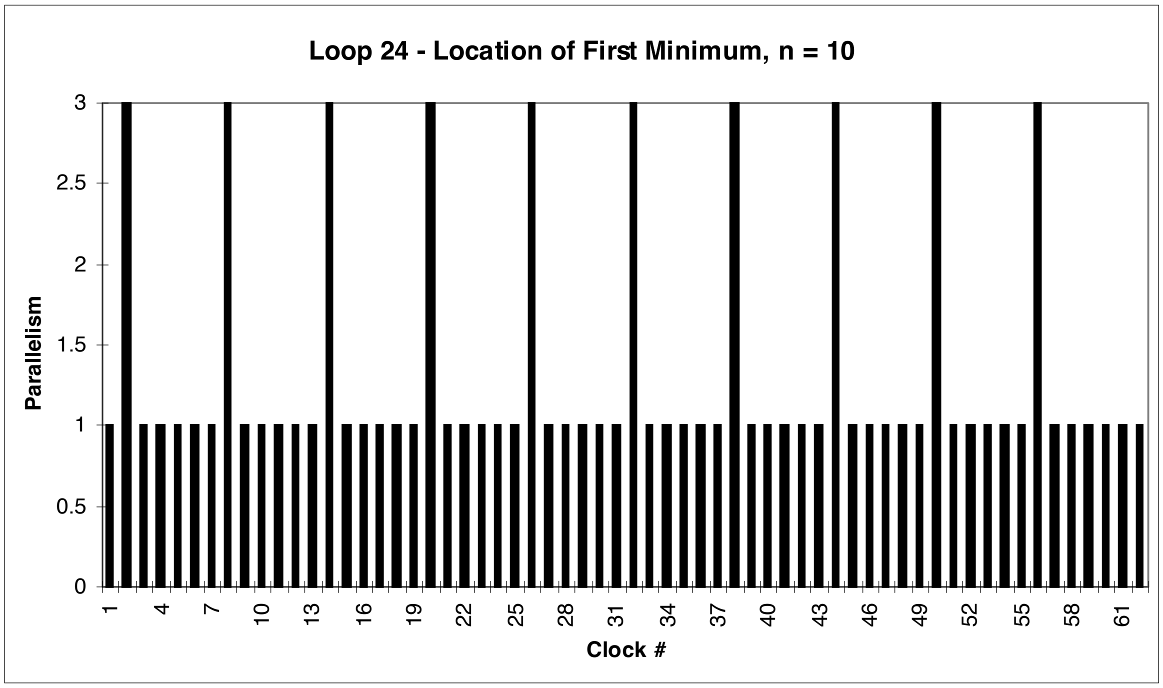

Loop 21 - Matrix Multiplication, n = 10
Incoming
Outgoing
4000
3500
3000
2500
2000
1500
1000
500
0
# of Arcs
1
2
3
4
5
6
7
8
9
10
11
12
Level #


Loop 21 - Matrix Multiplication, n = 10
1200
1000
800
600
400
200
0
Parallelism
1
2
3
4
5
6
7
8
9
10
11
12
Clock #


Loop 22 - Planckian Distribution, n = 101
Incoming
Outgoing
700
600
500
400
300
200
100
0
# of Arcs
1
2
3
4
5
6
7
8
9
10
11
12
13
Level #


Loop 22 - Planckian Distribution, n = 101
350
300
250
200
150
100
50
0
Parallelism
1
2
3
4
5
6
7
8
9
10
11
12
13
Clock #


Loop 24 - Location of First Minimum, n = 10
Incoming
Outgoing
# of Arcs
Level #


Loop 24 - Location of First Minimum, n = 10
Parallelism
Clock #

Appendix E
Source Code and Executable Programs

The IF1 compiler, viewer, and parallelism analysis tools consist of approximately 10,000 lines of source code. Due to large size, the source code was not included in this thesis. The source code and the executable programs for both Microsoft Windows version and X Windows version may be obtained by contacting Dr. Mitchell A. Thornton.

Dr. Mitchell A. Thornton

Department of Computer System Engineering

313 Engineering Hall

University of Arkansas

Fayetteville, AR 72703

E-mail: mat1@engr.uark.edu