A FACILITY FOR THE DISPLAY AND MANIPULATION OF DEPENDENCY GRAPHS

Ramachandran Sriram

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-22 June 1988

A FACILITY FOR THE DISPLAY AND MANIPULATION OF DEPENDENCY GRAPHS

by

RAMACHANDRAN SRIRAM, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May, 1988

Acknowledgments

I wish to thank Dr.J.C. Browne for the guidance and support he provided during my research work. Despite his busy schedule, he steered my work in the right direction and always found time to advise me. Prof. Harvey Cragon carefully reviewed my thesis. My work with Stephen Sobek was a learning experience in itself. Easwar provided much of the input that was used to test my software.

My parents have been a constant source of encouragement and inspiration. But for them, all this would not have been possible.

RAMACHANDRAN SRIRAM

The University of Texas at Austin

May, 1988

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Goals

The increasing availability of parallel architectures has prompted research towards the formulation of new paradigms for parallel programming. Conventional programming languages and their extensions have been found to be deficient in addressing the inherent complexities introduced by parallelism in architectures and algorithms.

In addition to a description of the modules that are to be executed concurrently, the specification of a parallel computation must explicitly state the synchronization constraints and communication between the modules. Thereby, the programmer ensures that the program executes according to the specification though it might follow different execution paths on each invocation. Programmers have become adept at specifying the sequential units in isolation. A new representation medium is required to aid the programmer in expressing parallel programs on the whole. The availability of low-cost workstations with bit-mapped graphics capabilities has prompted the use of graphical languages as a natural medium for parallel programming [SOB 86,SOB 88,DAV 82,PONG 86].

In such a programming framework, a directed graph is used to represent a computation. The nodes represent sequential modules and the arcs specify the dependency relations between these modules. Each module must have a firing rule which must be satisfied before the execution of that particular node. The Unified Computation Graph model proposed by Browne [BRO 85,BRO 86] decouples the sequential units of computation from the parallel structure that dictates their execution. This separation focuses the attention of the programmer on parallel structures and presents a more intuitive view of the parallel computation. Also, the representation of the parallel structure is at a level which alleviates the need to incorporate execution environment details. Computation graphs are hierarchical. Each node can be a subgraph, thereby providing the programmer with a powerful tool for controlling complexity.

Dependency graph representations have been found to be extremely powerful in addressing a variety of related problems. They have been used as a basis for the efficient mapping of algorithms to real architectures [KIM 87]. They can be viewed as a front-end for the translation of specifications into special-purpose VLSI architectures [DES 88]. Dependency graph representations of existing sequential programs can be extracted and used for parallel structuring [EAS 88]. The modularization of the sequential units of computation enhances the re-useability of these software components [LEE 87].

A general facility is needed for the display and manipulation of the dependency graphs in these research domains. Such a tool must be extendable to provide all the functionality required in a specific area while preserving generality. This thesis describes the development of the Interactive Dependency Graph Analyzer (*IDeA*), a graphical tool that meets these requirements.

1.2 Approach

It was recognized that IDeA must provide an extensive set of display operations, combined with a set of domain-specific functions.

Dependency graphs can be arbitrarily large depending on the size of the computation and on the level of abstraction. Therefore, to avoid the appearance of "spaghetti" graphs, an automatic layout algorithm is incorporated into the environment. IDeA provides a set of browsing and viewing operations to enable the user to view the entire graph or portions of it in a convenient manner.

Typically, the dependency graph representations could be stored in files or in databases and hence no assumptions on the input are made apriori. Each node has a generic pointer (database record, file name and line number etc) and can represent entire programs, subroutines or single statements. IDeA had a built-in notion of subgraphs thus enabling hierarchical representations in the dependency graph. IDeA incorporates functions for collapse of subgraphs into nodes and expansion of nodes into subgraphs.

A related project [EAS 88] is aimed at the extraction of call graph, data flow and control flow graph representations of sequential Fortran programs. This project provides an ideal test-bed for studying the strengths and weaknesses of IDeA in a real environment. The notion of hierarchy is demonstrated by allowing the user to collapse call graphs in a bottom-up manner.

IDeA is envisioned as an interface to the Computation Oriented Display Environment(CODE) [SOB 86]. Figure 1.1 shows the proposed organization.

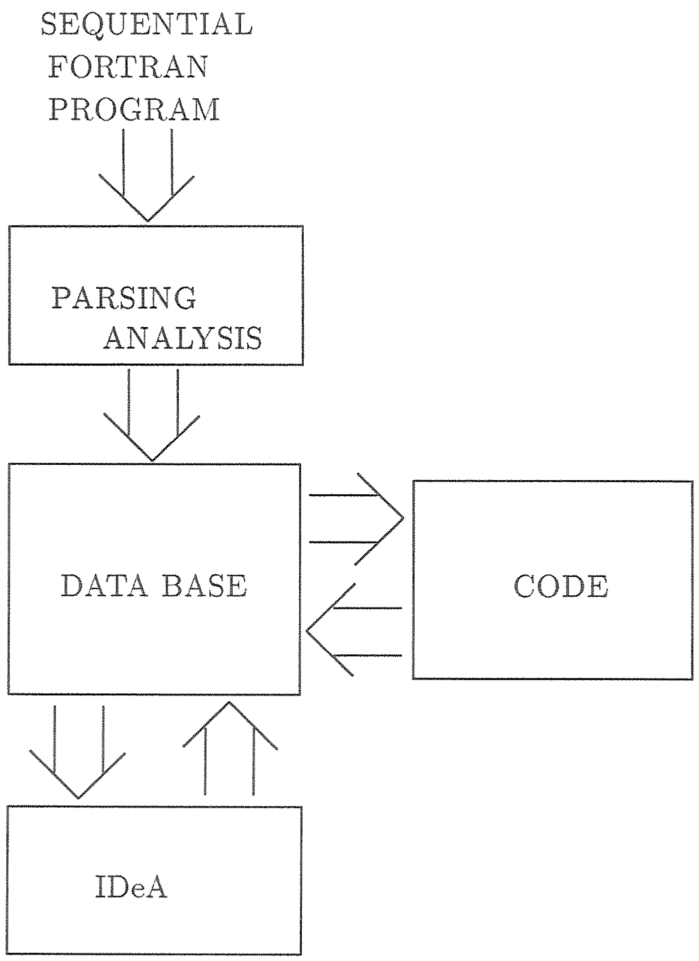


Figure 1.1: The CODE - IDeA interface.

A sequential Fortran program is parsed and the results of the analysis are stored in a relational data base. The interface between IDeA and the data base will guide the user in extracting a computation graph that can be understood and executed by CODE.

1.3 Related Work

This section describes related work in the areas of graph-oriented computation models and languages, visual/graphical programming, graph layout/browsing tools and analysis of Fortran programs for parallel structuring.

The formulation of a parallel computation as suggested by Browne [BRO 85,BRO 86] is the underlying computation model assumed in this thesis. This model can be used to describe algorithms and languages. The Computation Structures Language (*CSL*) is based on this model and was developed in the context of the Texas reconfigurable Array Computer. CSL is a graph language for specifying the configuration and traversal of a computation graph.

An alternate computation model suggested by [ADA 68] is based on data flow. Data flow models are based on tokens flowing from one node to another, where these tokens are instantiations of data objects. Data flow program graphs have also been suggested by Davis and Keller [DAV 82]. They extend the notion of tokens by allowing data structures to be constructed on the arcs. The notion of data flowing from one node to the other suggests the usage of a graphical representation as a logical and intuitive specification of the model.

CODE (Computation Oriented Display Environment), is a graphical parallel programming environment [SOB 86,SOB 88,BRO 85]. CODE is an architecture and language independent programming environment based on the unified model of computation described earlier. In using CODE, the programmer draws a dependency graph representation of the parallel computation which is then translated for a specific target architecture. The layout and hierarchical capabilities of IDeA suggest that it could provide a front-end for CODE in handling dependency graphs that are extracted from existing sequential programs.

GPSM is a graphical environment for programming simulation models [BDN 86]. Pong describes *I-PIGS*, a programming environment designed to support concurrent programming [PONG 86]. I-PIGS allows the programmer to create and edit processes, ports, communication links and control constructs directly on the screen.

GRAB, a general purpose graph layout and browser system is described by Rowe [ROWE 86]. IDeA uses a hierarchical layout which is essentially based on the GRAB algorithm. Implementation modifications were made to this algorithm to produces reasonable layouts with minimum turnaround times.

The conversion of large sequential Fortran programs to a parallel form has been investigated widely [AllK 82,KKLP 81]. Research efforts have been directed mainly towards the study of parallelism in DO-Loops, justified by the frequency of occurrence of such loops in real Fortran programs. *PTOOL* is a semi-automatic system that performs interprocedural analysis and uses a procedure call as the basic the basic unit of parallelism in its model [ABKP 85]. PTOOL allows the programmer to select regions for parallel execution and analyses dependencies to detect potential conflicts in the programmer's specification.

1.4 Organization

Chapter 2 discusses the model of parallel computation underlying this thesis. Chapter 3 describes the implementation design, the layout and the interface. The application of IDeA in Fortran call graph representations is described in Chapter 4, and the results and conclusions are summarized. Appendix A is a users manual.

Chapter 2

Model of Computation

This chapter describes the fundamental model of computation underlying IDeA. Graphical representations of procedural parallel programs and data flow programs are studied within the framework of IDeA.

Browne [BRO 85] defines a model of computation as :

1. Descriptions of primitive data types and operations on these data types.
2. Composition rules for creating the Schedulable Units of Computation (SUC's) from the primitive data types and operators.
3. Formulation for the creation of name spaces in which the SUC's execute.
4. Specification of the relations between the schedulable units of computation. These include the granularity of synchronization, constraint relations, the type specification of data objects that traverse the dependencies, and functional relations between the dependencies.

IDeA uses the Unified Computation Graph [BRO 85,BRO 86] as its underlying model of computation.

2.1 Unified Computation Graph

The Unified Computation Graph (UCG) is a framework for the formulation and representation of parallel computations. The computation graph is a directed graph in which the nodes are Schedulable Units of Computation (SUC's) and the edges represent dependency relations between the modules. A computation is specified by the traversal of such a graph.

A Schedulable Unit of Computation is defined by at least one initial state, a sequence of active states and at least one final state. The initial state includes the persistent internal state, and the binding of values from the input arcs to uninitialised names in the SUC. The final state specifies the assignment of values to the output arcs. The specification of a SUC makes no assumption about the execution granularity. Therefore, SUC's may represent individual statements, procedures or entire programs.

Any node in the computation graph can be a subgraph, thereby allowing hierarchical resolution of the nodes. Subgraphs are extremely powerful in controlling the complexity of the representation. IDeA is hierarchical and allows the creation of subgraphs in its representations.

IDeA supports schedulable units of computation by providing attribute lists for each of the nodes in the graph. Each node has an unique name and may be indexed to support parameterization. The attribute list in IDeA may be extended for domain specific representations. For example, it may be required to assign weights to each of the SUC's if the relative execution times of the SUC's are needed.

The arcs in the computation graph represent dependency relations between the SUC's. Dependencies specify either a synchronization relationship or a producer/consumer relationship. Synchronization relations may enforce ordering constraints or mutual exclusion. IDeA generalizes the specification of dependencies by providing attribute lists for the edges in the graph. The attributes must describe the *type* of the dependency. Data dependencies have associated data types to describe the data objects that will traverse that edge.

The UCG model allows for functional relationships between dependency relations. Thus, sets of arcs entering or leaving a node may be joined by *and* or *or* operations. CODE defines a special node type called a *filter* to implement these firing rule constraints. IDeA does not implement this special type. Therefore, such firing rules have to be synthesised by the creation of new SUC's.

2.2 Procedural Programming

Procedural programming languages are well represented within the framework of the Unified Computation Graph model. The schedulable units of computation maybe procedures in a sequential programming language. The input parameters to these procedures are specified as data dependencies. Control dependencies are used to specify additional synchronization and mutual exclusion constraints between the procedures that can execute in parallel. The Computation Structures Language (CSL)[McSHEA 86], and the Computation Oriented Display Environment (CODE) [SOB 86] are parallel programming languages based on the UCG model.

The parallel structuring of sequential Fortran programs has been widely investigated [AllK 82,ABKP 85,KKLP 81]. Most of these research efforts have been directed at extracting parallelism at the DO-loop level. A procedural programming model suggests that the granularity of execution must be at the procedure (function) level. In particular, procedure calls that occur inside DO loops offer much potential for large granularity parallel structuring.

A *Call graph* is a directed graph representation of the control flow relationships among the procedures of a program. Each node of the graph corresponds to a procedure and each arc represents one or more *calls* (or references) from the calling procedure to the called procedure. Call graphs represent procedure calls only and do not explicitly represent the return from the calls.

Each edge in the call graph can be viewed as a control dependency that implicitly specifies the following:

- A sequencing between the calling procedure and the called procedure.
- A sequencing between the called procedure and the segment of the calling procedure *following* the call.

Also, such a *CALL* control dependency has associated with it the parameter list for the call (the list may include global variables that are modified within the called procedure). The parameter list can be decomposed into a set of data dependencies in the computation graph. If values have to be returned to the calling procedure, they are represented as data dependencies. The return from the call is a pure control dependency if the called procedure does not have any return values. This control dependency may be converted to a data dependency, if required [AKPW 83]. The sequencing between successive calls may be ensured by additional control dependencies (and perhaps, by the use of mutual exclusion dependencies). The functional relationship between the set of incoming arcs in this representation is the *and* operator. The nodes will fire when there exist tokens (parameters) in all the incoming arcs.

Static variables inside the procedures represent the persistent internal state of the SUC's. The execution of the procedure specifies the sequence of active states of that SUC. Hierarchical representations of the call graph are specified by collapsing a node (procedure), and all the nodes it calls, into a subgraph. Chapter 4 describes the call graph collapse in greater detail.

IDeA has been used to display call graphs extracted from sequential Fortran programs. The generality offered by IDeA may be exploited to aid in the extraction of computation graphs from the call graph representations. It is envisioned that IDeA will provide an interface to CODE, such that a user may interactively convert a sequential program to a form that can be mapped into the CODE programming environment.

2.3 Data Flow Program Graphs

Davis [DAV 82] defines a *data flow language* as an applicative language based on the notion of data flowing from one function entity to another. Applicative languages are extremely modular and this enables the clear decoupling of the execution of independent modules in the programs. Graphical representations are extremely useful in the specification of data flow program graphs.

Data flow graphs have firing rules associated with the nodes. A node can fire when it receives its input tokens. A graphical representation of a data flow graph specifies the interaction between the nodes and the interconnecting arcs in a manner similar to the UCG.

Data flow programs can be *composed* into larger programs due to the modular structure. The connectivity of such modular compositions can be represented in a graphical manner.

Graphs can be used to attribute a formal meaning to a program. An operational definition specifies a permissible sequence of operations during the execution of the program. A functional description encompasses the entire program as one single function and is independent of the execution model.

It was mentioned earlier that the Unified Computation Graph model defines parallel computations as declarative hierarchies. Similarly, the hierarchical resolution of data flow graphs is done by *macroexpansion* [DAV 82]. A macrofunction is defined by specifying a name and associating it with a graph, in a manner similar to the use of subroutines and procedures in conventional programs. The macroexpansion in data flow programs may be viewed as a run-time operation. A node in the data flow program graph can, in effect, be replaced by its macro definition during the execution of the program.

Data flow program graphs can be represented by using IDeA. The data flow model is a specific instance of the UCG model. The specification of a data flow graph must additionally include the granularity of the SUC's, data/demand control, token/stream representation of data and static or dynamic graph structure. [BRO 85].

Chapter 3

Implementation

This chapter describes the layout algorithm, the underlying graphics model, overall design and extensibility of the system.

3.1 Layout

The layout algorithm used by IDeA is a derivative of the hierarchical layout algorithm developed by Sugiyama et. al. [SUG 84], and the modified version used in the GRAB project [ROWE 86]. Figures 3.1 and 3.2 show graphs displayed by IDeA.

The first phase of the layout algorithm detects all the edges that complete cycles and marks them as being *cycle edges*. This phase is compute intensive, and can be disabled if the user has apriori knowledge that the graph contains no cycles. The second phase assigns the nodes to levels. A node is assigned to the current level if it has no incoming edges, or if all its parents have already been assigned. The incoming edges that have been marked as cycle edges are ignored in the levelling phase.

Long edges are edges that span across levels, and are broken up by the introduction of dummy nodes in the intermediate levels. It is to be noted that *long cycle edges* are also broken up in this phase. These dummy nodes have no semantic meaning, but are treated by the layout algorithm as ordinary nodes. The introduction of the dummy nodes and edges ensures that all edges connect nodes that are separated utmost by one level.

Figure 3.1 shows a broken long edge *[sort,print]*, and a broken long cycle edge *[index,sort]*.

These dummy nodes and dummy edges provide an additional measure of flexibility in positioning the long edges such that the overall number of edge crossings is reduced. The introduction of the dummy nodes (and the creation of subgraphs as described later) necessitated the inclusion of a global record of the true origin and the true destination of the edges.

IDeA has a built in notion of a *current graph* (or current subgraph). The current graph is defined by the current root and the different levels in the graph. This implicit assumption enforces the severe restriction that any graph that can be displayed must have one root (thereby making the graph at the highest level a subgraph in itself). If multiple roots are detected, the user is prompted to choose a root. In the current implementation a decision was made to retain the invisible portions of the layout in the data structure if multiple root nodes are detected. While this approach provides the user with an option to switch between "different" graphs, the quality of the layouts tend to deteriorate because large portions of the overall layout are now invisible. However, the design decision is reasonable since we do not expect to be frequently presented with multiple root graphs.

Figure 3.3 shows the layout of the graph at this stage of the algorithm. A root node has been identified, all long edges have been broken up and the nodes have been levelled. The horizontal skewing in figure 3.3 is due to the presence of invisible multiple roots. The following phases determine the positioning of the nodes within each of the levels.

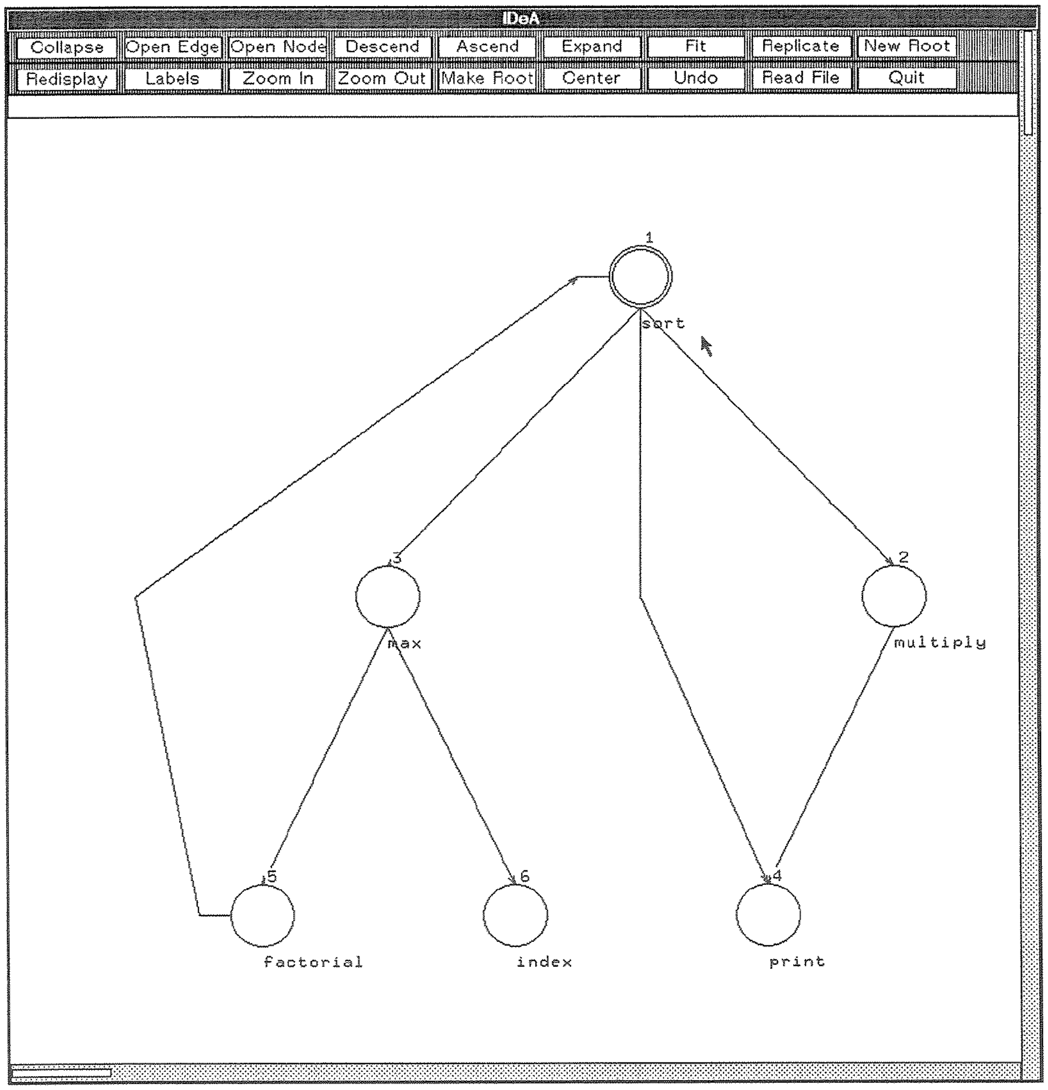


Figure 3.1: A graph drawn by IdeA.

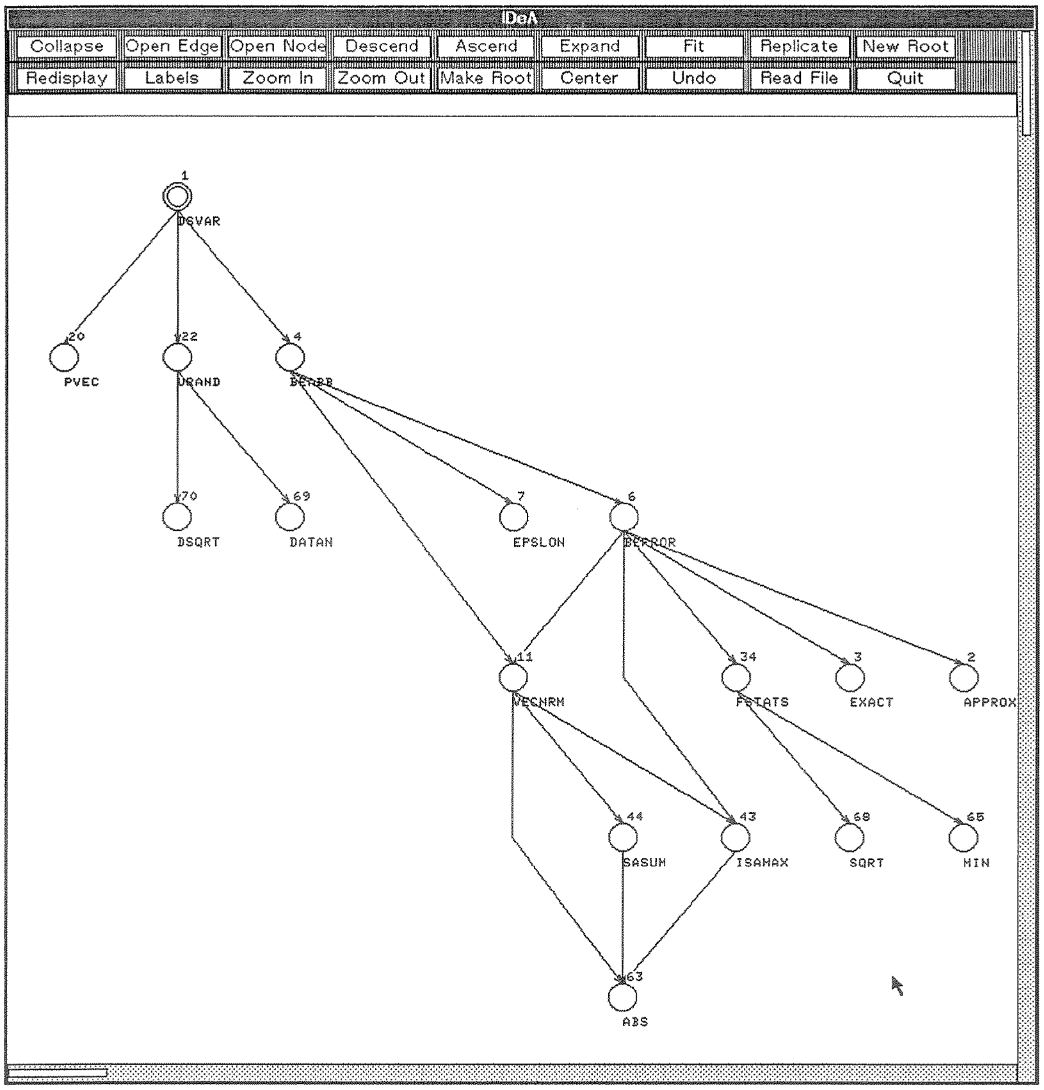


Figure 3.2: Call-graph drawn by IdeA.

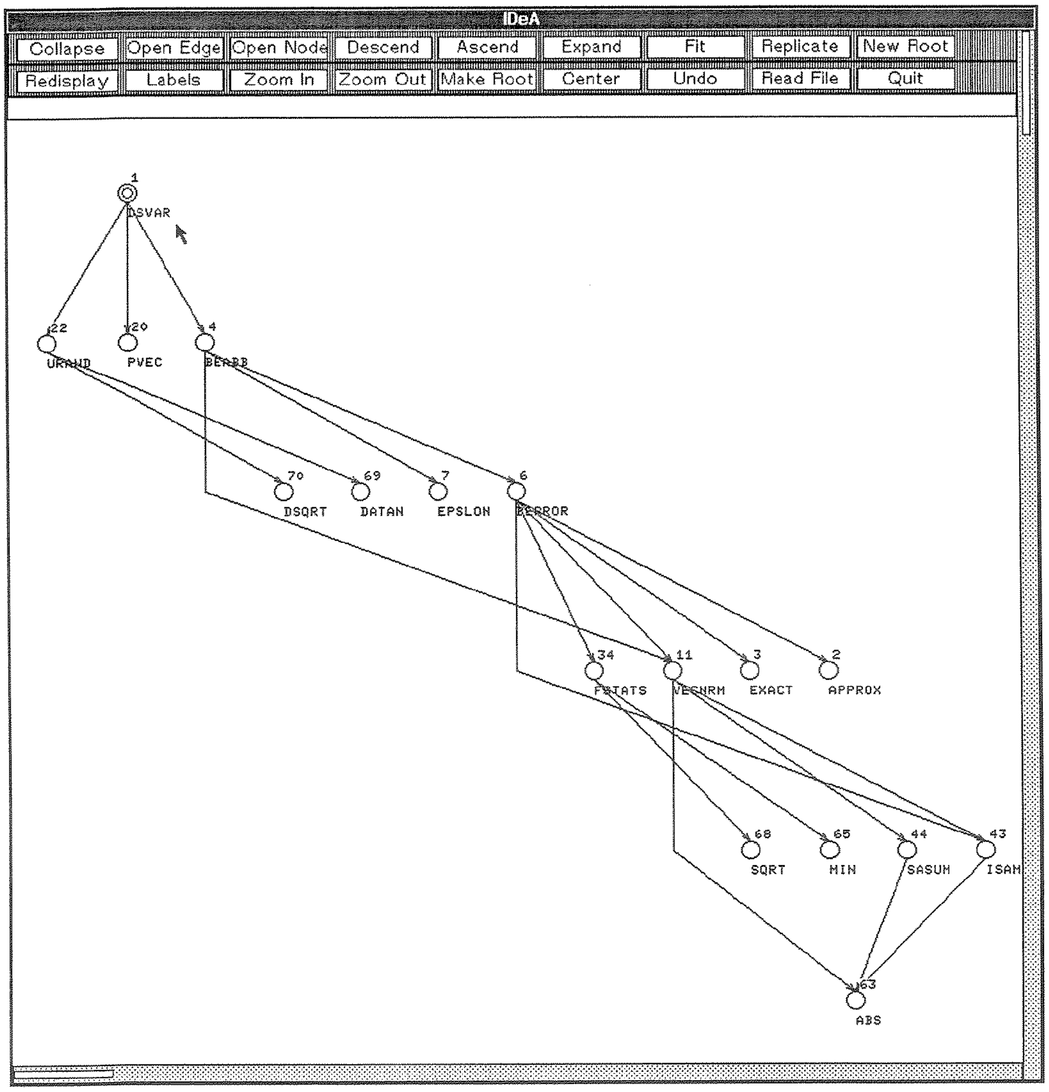


Figure 3.3: Call-graph after the levelling phase.

The horizontal positioning of the nodes in a level is based on a measure called the *barycenter* [ROWE 86]. Each node is assigned an *up barycenter* which is the average position of all its immediate predecessors, and a *down barycenter* which is the average position of its immediate successors. Dummy nodes are included in the barycenter calculations.

The algorithm makes a downward pass over the graph, sorting the nodes in each level by their up barycenters. This phase attempts to minimize the edge crossings with respect to the predecessors of each of the nodes. Figure 3.4 shows the layout after the nodes have been sorted by up barycenters. Similarly, an upward pass is made through the graph to sort the nodes with respect to down barycenters. Figure 3.5 shows the graph after it has been sorted by down barycenters. It is to be observed that the position of some of the nodes may swing from one extreme to the other during these sorting phases. Therefore, the subsequent sorting phases sort based on the average of the up and the down barycenters. The algorithm can be made to terminate either after a fixed number of such sorting phases, or when the edge crossings fall below a threshold. The calculation of the number of edge crossings after every phase is computationally expensive and has been avoided. Also, since IDeA incorporates subgraphs, it is expected that the number of nodes will typically be small enough to be displayed with a fixed number of passes. IDeA makes two such sorting passes and the resultant graphs have been reasonable.

The last phase of the algorithm assigns the actual coordinates to the nodes. Collisions at a particular coordinate in the horizontal positioning are resolved on a FIFO basis. A minimum horizontal grid spacing is enforced between the nodes to make the graphs more readable. The vertical spacing between the levels is adjusted to meet a minimum slope requirement for the

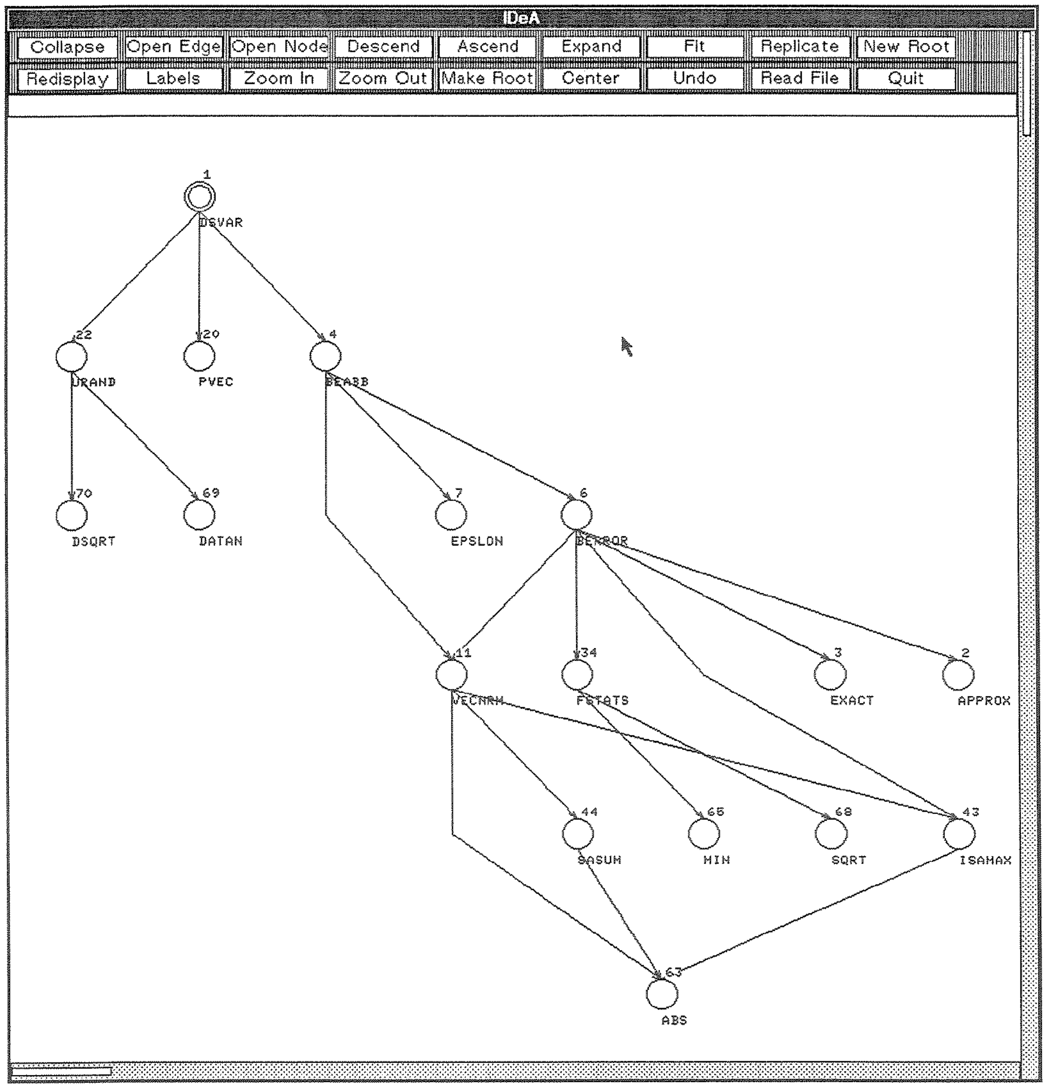


Figure 3.4: Call-graph after sorting with respect to predecessors.

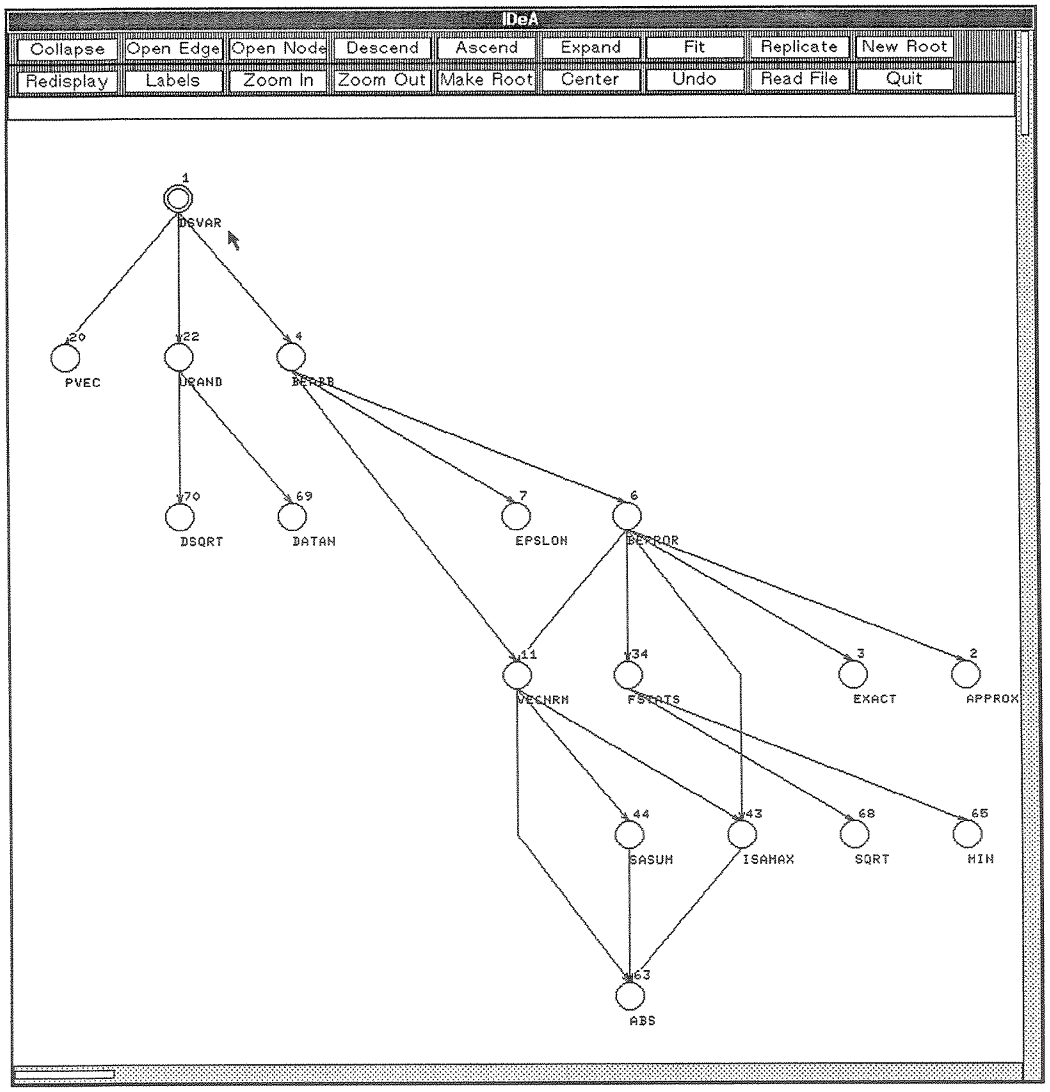


Figure 3.5: Call-graph after sorting with respect to successors.

edges that connect the levels. This restriction prevents the appearance of nearly horizontal edges in the graph. Figure 3.2 shows the final layout after the call-graph has been processed entirely.

3.2 Graphics model

IDeA uses a graphics model based on the classical notion of a *window/viewport* pair [FOL 84]. A *window* is a rectangular portion of the coordinate system in which the graph description has been stored. A *viewport* is the rectangular portion of the screen onto which the window and therefore the window contents are to be mapped. A transformation is defined to map coordinates from the window to the viewport.

Initially, IDeA defines the window to encompass the entire graph structure and then calculates the scaling factors for the transformation. IDeA always attempts to fit the entire graph in one screen-full. If the scaling factors are extremely small (due to large windows), the scaling factors are restricted to a pre-determined lower bound. Therefore, IDeA ensures that the graph will always be displayed with reasonable dimensions.

As stated earlier, IDeA maintains a current root. Window calculations are done on the graph rooted at the current root. Therefore, the graph is automatically scaled whenever the current root is changed.

IDeA does not maintain conventional display lists, but stores the screen coordinates of the nodes to avoid unnecessary computation. IDeA has the notion of a *current display mode.* The SCALE_MODE mode is used when the current window size has to be found and the scaling factors are to be set accordingly. The FIT command uses the SCALE_MODE to "fit" the graph on the screen. The REDISPLAY_MODE is used to indicate that the scaling factors and the screen coordinates have already been computed and the operation is simply to clear the screen and draw the graph again (as in the case of the drawing window being corrupted).

The SCROLL_MODE indicates that IDeA is to perform a scrolling operation. Scrolling is done very simply by offsetting the window according to the user's request without changing the size of the window itself. Therefore, the scaling factors remain the same, though the drawing origin is now different. Both horizontal and vertical scrolling are allowed. The CENTER command uses the scroll mode to center the graph on a chosen node.

The ZOOM_MODE indicates that the scaling factors have been modified. The ZOOM IN operation increases the current scaling factors and the ZOOM OUT operation decreases the scaling while keeping the window size a constant.

The X Window system automatically clips the output to remain within the boundary of the defined drawing surface [XLIB 86]. Therefore, IDeA does not make any attempt to keep track of the current clipping rectangle when portions of the window are visible in the viewport. A more sophisticated clipping mechanism will prevent the graphics routines from traversing the invisible portions of the graph.

The graphics model does not accommodate multiple viewports. The the user is always presented with a single view of the graph.

The graphics model and the graphics routines are designed to be compatible with the hierarchical nature of the graph representation. Thus, any of the graphics operations can be invoked at any stage of the graph display.

3.3 Design and Extensibility

3.3.1 Object Types

IDeA defines three types of nodes:

1. Ordinary nodes (Schedulable Units of Computation).
2. Subgraph nodes.
3. Subgraph entry nodes (to enforce single entry into subgraphs).

A status bit indicates the node type and a programmer can easily define his own node type by setting a new status bit.

Ordinary nodes usually have some associated code fragment. The nodes store the code pointer as a file name and a line number. This can be extended to indicate a data base record if IDeA is interfaced to a data base.

In the current implementation, IDeA allows one generic edge type. Edges can be extended to include type information by the addition of a status field. IDeA keeps track of the *true start* and the *true end* of an edge. This information is useful to track edges that traverse subgraphs, and long edges that are broken into a number of dummy edges. Edges may have associated property lists.

The current graph structure stores the current root of the graph and pointers to the different levels. Therefore, the graph can be accessed either by a depth first traversal, or more efficiently by traversing the level lists if hierarchy is not required. The current window extent and the scaling factors are also stored in the current graph description.

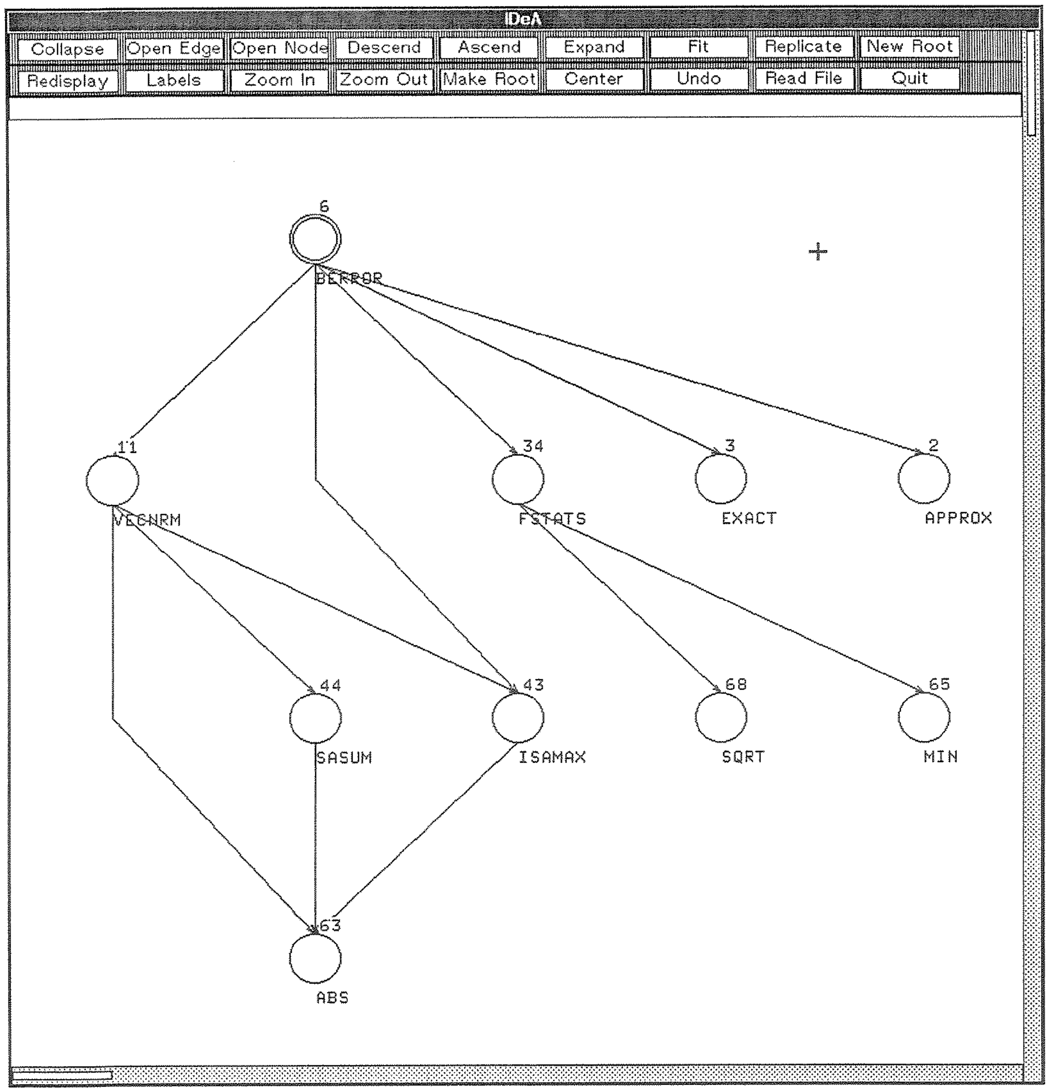


Figure 3.6: Call graph with a new root

3.3.2 Functions

The current root is always highlighted with a double border. The current root can be changed by using the MAKE ROOT command. This command is extremely useful in isolating logical portions of the graph (as opposed to zooming which isolates geometric portions of the graph). For example, in a call-graph representation, the MAKE ROOT command can be used to isolate a particular module and all other modules that it calls. Figure 3.6 shows the call-graph after the MAKE ROOT command has been invoked on module *BERROR* in the original call-graph shown in Figure 3.2.

The COLLAPSE operation is used for the creation of subgraphs. Any *ordinary* node and its descendants can be collapsed into a subgraph. If there exist edges that begin at nodes not included in the subgraph, but terminating in nodes that are to be included in the subgraph, IDeA enforces single entry into the subgraph by the creation of a single entry node. If the node *BERROR* in Figure 3.2 is to be collapsed, *[BEABB, VECNRM]* is an *external edge* and will result in a graph as shown in Figure 3.7. If external edges are detected in a COLLAPSE operation, the layout algorithm will be invoked to determine the position of the new subgraph. The newly created subgraph is highlighted each time the layout algorithm is invoked so that the overall context is not lost. If external edges are not present, the newly created subgraph node will simply replace the original node that was collapsed. The issues to be addressed in resolving such external edges are discussed in the context of call-graph collapse in the next chapter.

The DESCEND operation allows the user to descend down the hierarchy into the chosen subgraph. Figure 3.8 shows the result of descending into the subgraph *BERROR* in Figure 3.7. The single entry nodes have

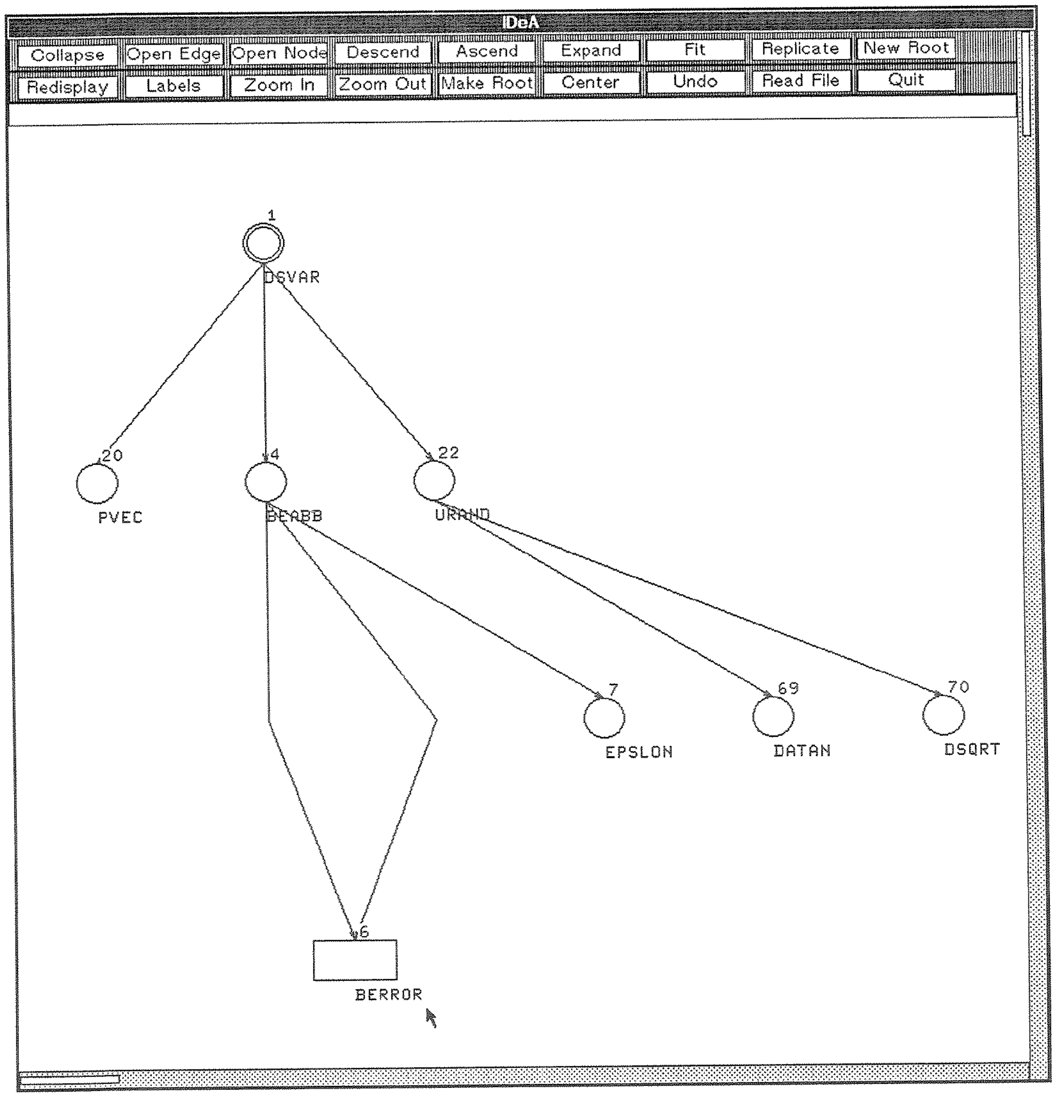


Figure 3.7: Call graph after subgraph creation

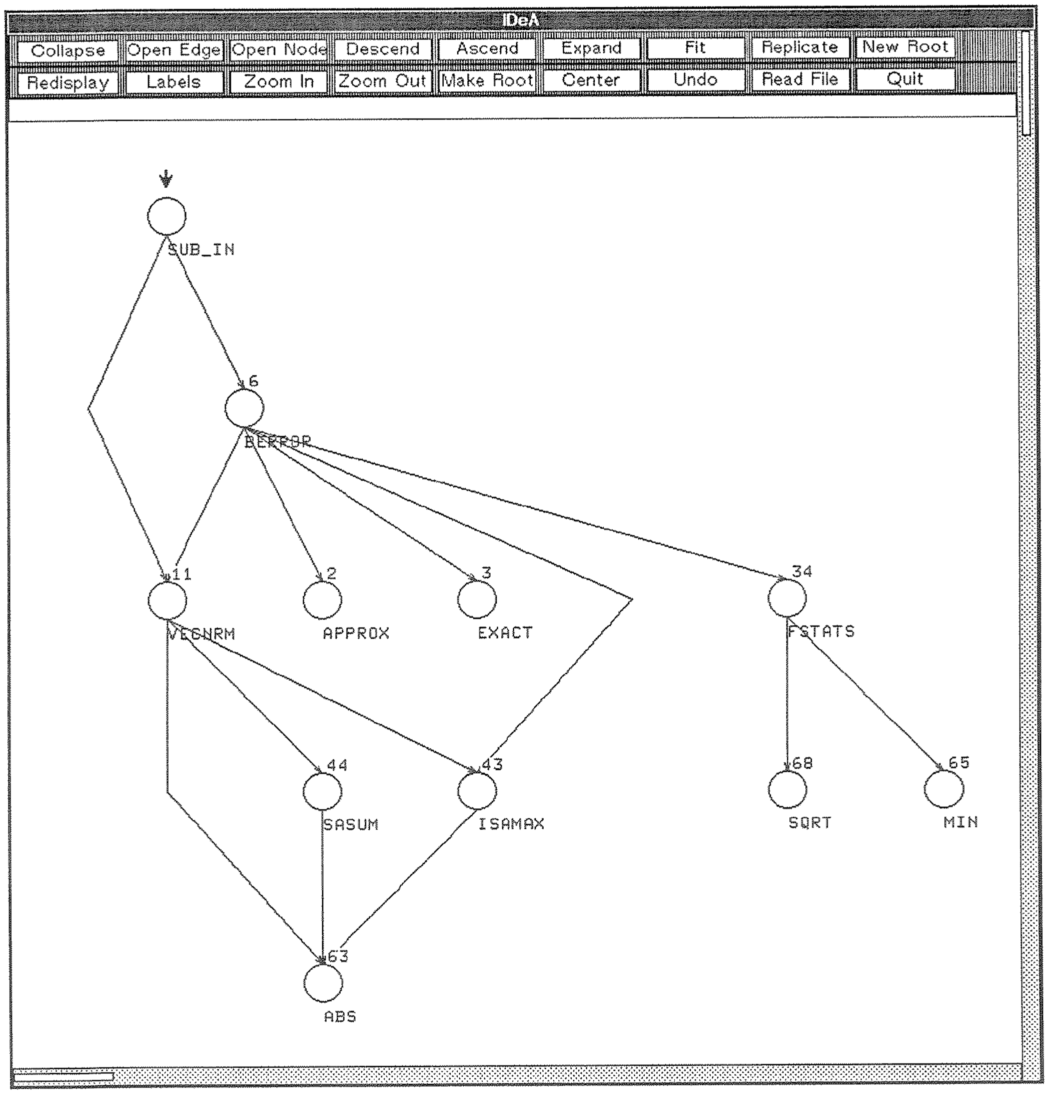


Figure 3.8: A subgraph created by IDeA

no associated code fragment and cannot be collapsed into subgraphs. The OPEN EDGE command will display the true origin and the true destination when invoked on an external edge in a subgraph.

The ASCEND operation allows the user to move up the hierarchy. The EXPAND operation breaks up a subgraph and inserts all the nodes of the subgraph in the current graph. Expanding the subgraph *BERROR* in Figure 3.7 will yield the original graph shown in Figure 3.2. The EXPAND operation always invokes the layout algorithm.

The REPLICATE operation can be applied to *ordinary leaf nodes.* This operation is useful in the parameterization of the nodes. It prompts the user for lower and upper bounds on the indices.

Chapter 4

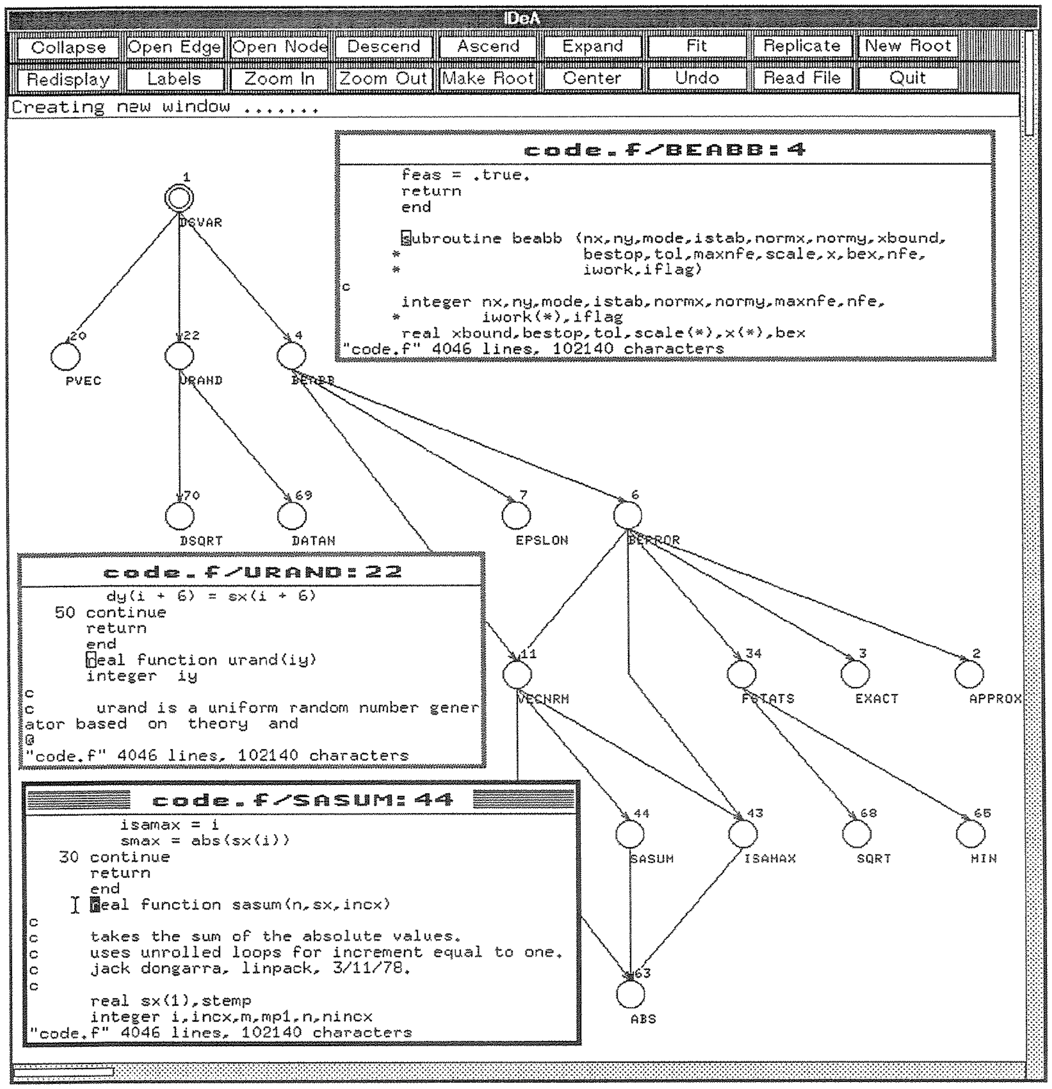
Results and Conclusions

Call graphs of existing Fortran programs were extracted in a related project [EAS 88]. The display and manipulation of these call graphs by the use of IDeA is described in this chapter. The results and conclusions are summarized.

4.1 Call Graph Representations

A Call Graph is a directed graph specifying the control flow relationships between the various subroutines in the program. As described in Chapter 2, one important procedural programming paradigm uses the subroutine call as the basic unit of parallelism. Therefore, call graphs were identified and used as an effective basis for interprocedural analysis.

The nodes in a call graph correspond to Fortran subroutines and the edges specify the calling relationships. A facility is provided in IDeA to enable the user to browse through and edit the code corresponding to each of the nodes. Figure 4.1 shows the invocation of the OPEN_NODE command on some of the nodes in the call graph. IDeA allows simultaneous viewing and editing operations on more than one node at a given time. This feature, while being extremely useful in perusing through multiple code fragments, introduces read/write conflicts on the code fragments. If this is found to be a



```
Creating new window .......
```

code.f/BEABB:4

```
      feas = .true.
      return
      end

      subroutine beabb (nx,ny,mode,istab,normx,normy,xbound,
     *                  bestop,tol,maxnfe,scale,x,bex,nfe,
     *                  iwork,iflag)
c
      integer nx,ny,mode,istab,normx,normy,maxnfe,nfe,
     *        iwork(*),iflag
      real xbound,bestop,tol,scale(*),x(*),bex
"code.f" 4046 lines, 102140 characters
```

code.f/URAND:22

```
        dy(i + 6) = sx(i + 6)
   50 continue
      return
      end
      real function urand(iy)
      integer  iy
c
c       urand is a uniform random number gener
ator based  on  theory  and
@
"code.f" 4046 lines, 102140 characters
```

code.f/SASUM:44

```
         isamax = i
         smax = abs(sx(i))
   30 continue
      return
      end
      real function sasum(n,sx,incx)
c
c     takes the sum of the absolute values.
c     uses unrolled loops for increment equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      real sx(1),stemp
      integer i,incx,m,mp1,n,nincx
"code.f" 4046 lines, 102140 characters
```

Figure 4.1: Multiple Edit windows in IDeA

problem, the editing operations will have to be confined to one window only. Also, provisions will have to be made to ensure reliable back-up of edited code fragments for effective source control.

Subroutines have well defined linkages, and therefore, call graphs can be used for hierarchical composition. IDeA takes a bottom-up approach in the hierarchical specification of the call graphs. If it is possible to identify a group of routines that can be executed independently, these routines can be collapsed into a single subgraph that can then be scheduled for execution. This collapse is typically done by identifying an independent subroutine and collapsing the subgraph rooted at this node in the call graph. However, the identification of these groups of subroutines in the call graph is complicated by the call-by-reference semantics of Fortran. Also, it is not atypical in real Fortran programs to have a set of "work-horse" routines that are called by several other subroutines. The simplistic collapse approach suggested above will not work well in this case because routines in a subgraph will have edges coming in from nodes that do not belong in the subgraph. Figure 4.2 clarifies the problem by an example.

In collapsing the call graph in Figure 4.2, it is assumed that the routines *SORT*, *MULTIPLY* and *VALIDATE* can be executed concurrently. The collapse of the nodes *VALIDATE*, *INPUT* and *CHECK* is therefore independent of the other routines in the graph as shown in Figure 4.3. To use the terminology of the Unified Computation Graph model, the subgraph *VALIDATE* in Figure 4.3 is now a Schedulable Unit of Computation and can be executed in parallel with other such units. However the similar collapse of the *MULTIPLY* routine is complicated by the fact that the *PRINT* routine is also called by the *SORT* routine. If it can be determined that the *PRINT*

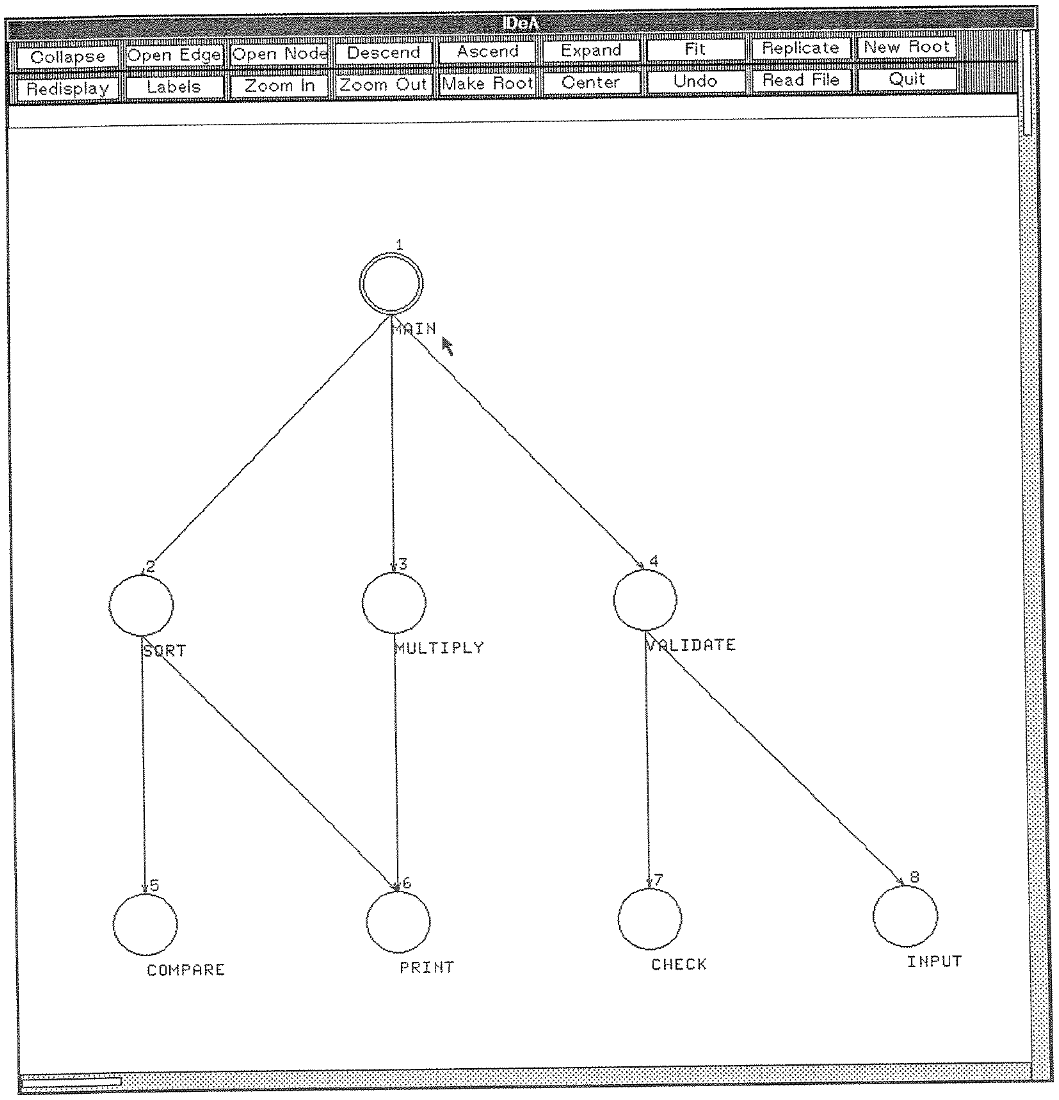


Figure 4.2: Subgraph collapse: Example 1

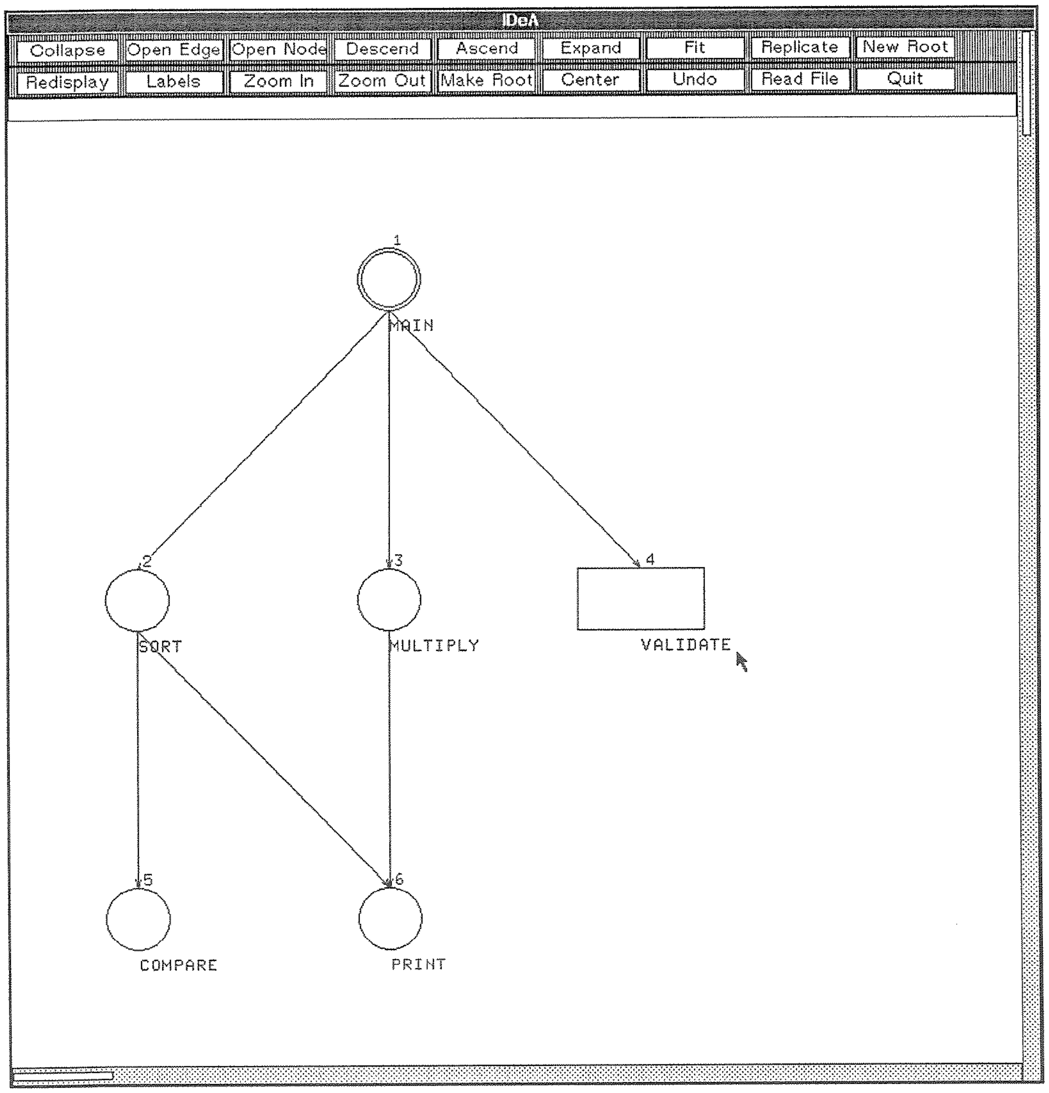


Figure 4.3: Subgraph Collapse: Example 2

routine has no persistent internal state, and if it does not affect any global variables, a copy of it can be made (by using the REPLICATE operator) and the collapse can proceed. The default COLLAPSE operation in IDeA does not make this assumption and will perform the collapse as shown in Figure 4.4. Single entry into the collapsed subgraph *MULTIPLY* has been enforced and the original edge *[SORT,PRINT]*, will have to traverse the subgraph *MULTIPLY*. Figure 4.5 shows the subgraph *MULTIPLY*. Single entry into the subgraph has been introduced by the addition of the SUB_IN node. The OPEN_EDGE operator shows the true origin of the broken-up edge in the message window. It is to be noted that the newly introduced node, SUB_IN, has no semantic meaning other than ensuring single entry. It cannot be operated upon by the COLLAPSE operator any further, and has no associated code fragment. The edge *[SUB_IN, MULTIPLY]* simply indicates that the *MULTIPLY* node is the node that was originally collapsed. By default, all subgraphs have the same name as the original node that was collapsed. If each of the subgraphs is to be packaged as a schedulable unit of computation, a mechanism is required to collect all the code fragments of the subgraph into one logical unit. In the current implementation, the user will have to bring up an edit window and perform this operation explicitly.

4.2 Conclusions

IDeA has established a framework for the display and manipulation of graphical representations of programs. The initial target was to support the conversion of sequential Fortran programs into parallel programs. The initial functionality reflects this target, but the functionality of IDeA can readily be expanded as knowledge concerning the use of graphically-oriented

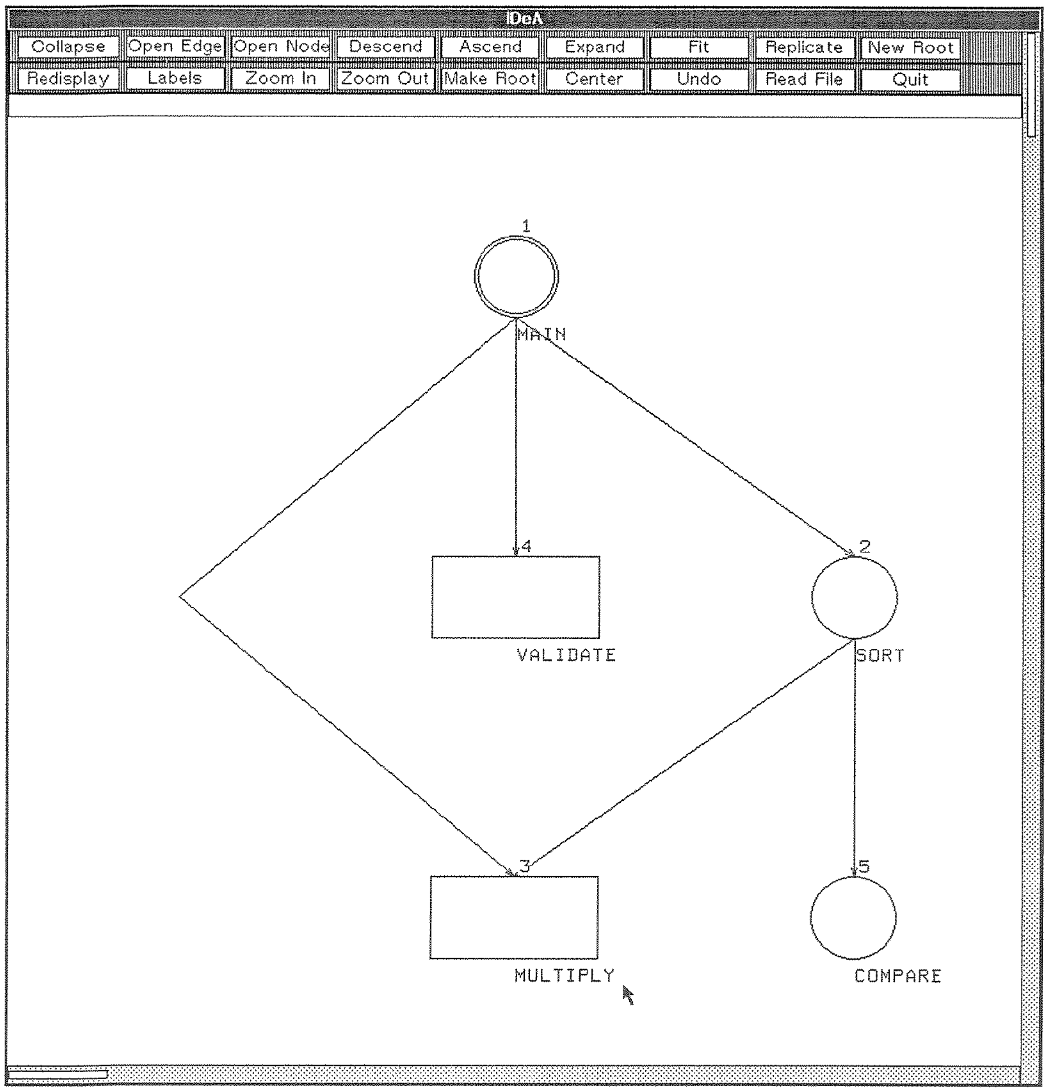


Figure 4.4: Subgraph Collapse: Example 3

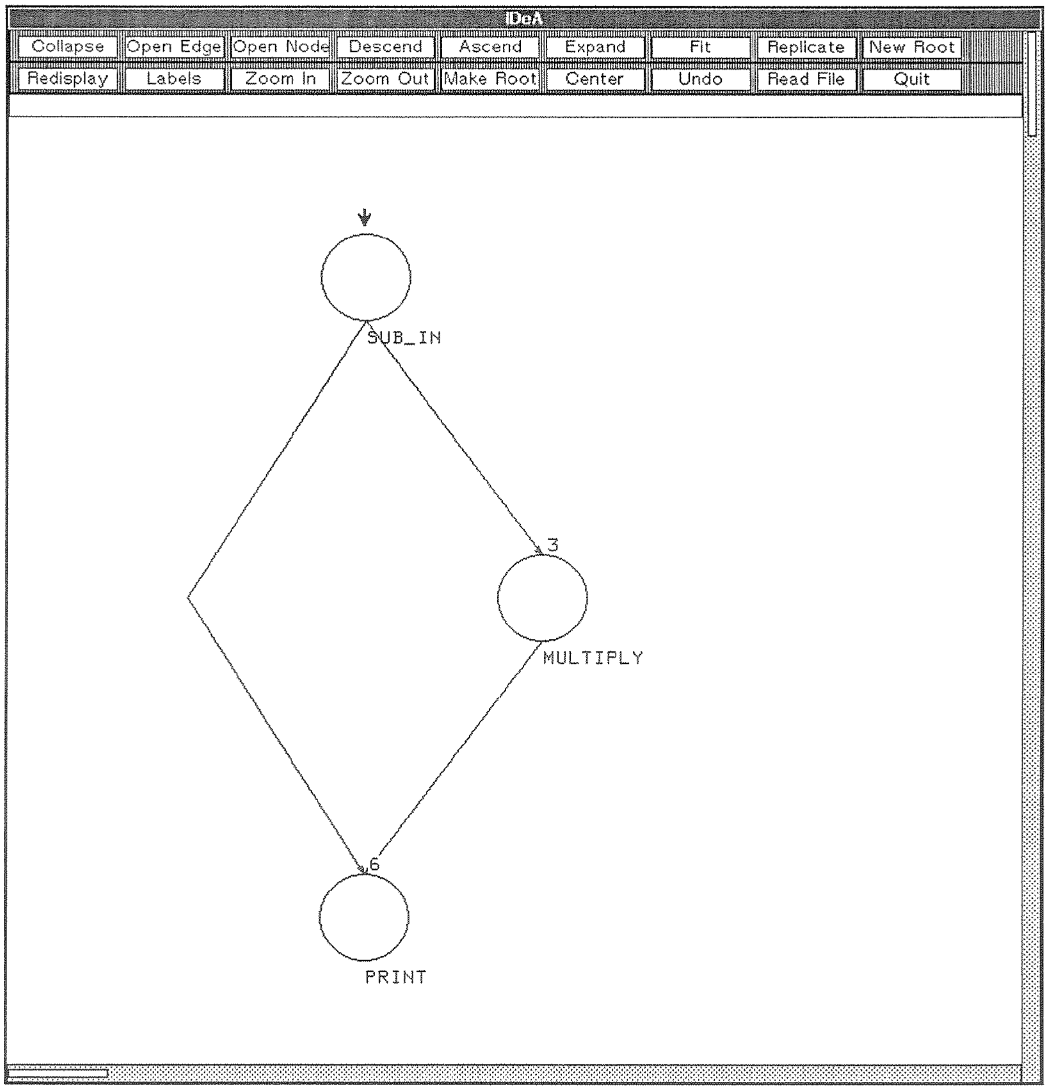


Figure 4.5: Subgraph Collapse: Example 4

methods for management of programs accumulates.

Hierarchical specification and resolution of computation graphs is supported by IDeA. In a parallel programming environment, the specification of computation graphs as declarative hierarchies is a powerful tool for controlling complexity.

The representation basis used by IDeA enables the clean separation between the Schedulable Units of Computations and the relationships between these modules. This representation will be invaluable, particularly as IDeA is enhanced to support a more detailed analysis of the generated computation graphs.

IDeA is a visual programming tool and provides the user with a more intuitive view of a parallel computation. It is equipped with an extensive set of operations for the display and manipulation of computation graphs. It provides an automatic layout for any given computation graph. The layout algorithm is fairly sophisticated and allows the display and manipulation of arbitrary data flow and control flow graphs. It makes provisions for handling cycles in the displayed graphs.

The effectiveness of IDeA for a real application was demonstrated by the display of call graphs derived from sequential Fortran programs. The bottom-up collapse of call graphs demonstrates the hierarchical capabilities that have been built into IDeA. IDeA is envisioned as a front-end to the Computation Oriented Display Environment. Therefore, future extensions to IDeA must concentrate on interfacing IDeA to a data base containing the computation graph descriptions. Such an interface will enable IDeA to be used for extracting a computation graph that can then be mapped to CODE.

Appendix A

User's Manual

A.1 Overview

This section describes the user interface principles underlying IDeA. IDeA is a menu-driven system, wherein the command selection is done by using a set of button menus. These button menus are located inside a menu window positioned at the top of the graphical interface. Any of the commands may be chosen by moving the cursor inside the appropriate button menu and clicking on a mouse button. Each of the command modes has a unique cursor to provide the user with a visual feedback of the current command mode. The messages generated by IDeA are displayed in the message window immediately below the menu window. If the message to be displayed is longer than the message window, it is printed on the window from which the program was invoked. The user's attention will be drawn to such a message by an appropriate message in the message window. All drawing operations will be performed within the large rectangular region called the drawing window. The vertical and horizontal windows bordering the drawing window are the scrollbar windows. The elevators inside the scrollbars will be positioned appropriately by IDeA. Text input is done by pop-up dialogue boxes. The text input in the dialogue box can proceed only if the mouse cursor is inside the dialogue box. IDeA automatically warps the mouse cursor into the dialogue box, each time a dialogue box is mapped on to the screen. However, if the user moves the cursor outside the dialogue box for some reason, the cursor must be re-positioned inside the box before the text input is resumed. The end of text input is signalled by a click on the "Ok" button. Confirmation of commands and user requests are done by the use of a pop-up notifier. Until the user clicks on the notifier and confirms or cancels the request, all activity will be frozen.

Some of the commands require the user to choose a node. The selection of a node can be done in two ways. The cursor can be moved inside the node to be selected and the left button of the mouse may be pressed to signal the selection. Alternatively, if the user desires to make the choice by using the node name (or node id) a click on the right button will bring up a dialogue box that allows the user to type in the name (or id) to make the required selection.

In the current implementation, resizing of the windows has been disabled. However, IDeA will refresh and redisplay all its windows when they are uncovered from underneath another window, or have otherwise been damaged. The root window can be moved and positioned anywhere on the screen. The command to move the window is dependent on the window manger currently in use.

A.2 Commands

A.2.1 Getting Started

IDeA can be invoked by moving into the appropriate directory and executing the command *idea*. In the current implementation, the input graphs to IDeA are stored in Unix files. The format of the input graphs is as follows:

```
<name of file containing the source code>
/begin_nodes/

<node name>  <node id> <line number>
.
.
.
.

/begin_edges/
<node id>,<node id>
.
.
.
.
.
```

In the current implementation, the source code for each of the nodes in the graph must be present in one file (a restriction imposed in the Fortran parsing stages). The first line in the input file must contain the name of this file. Each of the nodes in the graph must be given a unique name and a positive integer id. The line number specifies the starting line of the code for a particular node, so that the editor can be invoked to edit (or browse) through this code fragment. The edges in the graph are specified as node pairs.

A.2.2 File Commands

The READ FILE command creates a dialogue box and prompts the user for an input file name. If the specified file cannot be opened, an error message will be printed in the message window. If IDeA detects multiple roots at the highest level in the input, the user will be prompted to choose a root. A list of all the roots will be printed in the message window and the user will be prompted to make the selection through a dialogue box. IDeA always tries to scale the graph to fit inside one screen-full. However, if the graph is too large, IDeA will display only a portion of the graph so that the graph remains readable. The initial display of the graph will always contain the root node, so that the context is maintained. Also, the root node will be highlighted with a double edged border. After a graph has been read in, the READ FILE command can be executed again at any stage if the user wants to read another input file. IDeA will prompt for a confirmation of the request before destroying the currently displayed graph.

A.2.3 Display Commands

The REDISPLAY command clears the drawing window and displays the graph on the screen.

The LABELS command allows the user to turn on or turn off the display of the node names and the node ids. The selection is done through a set of button menus in a pop-up menu window. By default, the graphs will be drawn with only the node ids turned on. A selection through the LABELS command will be valid across different input graphs in a particular viewing session.

The ZOOM IN command scales up the display of the current graph by a constant factor. The ZOOM OUT command is the inverse of the ZOOM IN command - it scales down the display of the graph by the same constant factor.

The CENTER command allows the user to select and position a node at the center of the display area. This command allows the user to view a node and its context conveniently.

The FIT command scales the entire graph to fit in one screen-full. If the graph is too large, the FIT command enforces a minimum restriction on the scaling factors so that the graph is always readable.

The UNDO command is used to undo the last executed command. Most of the display commands can be undone.

Scrolling in the horizontal and vertical directions may be done by moving the cursor into the appropriate scroll bar and clicking on the middle button. The button click is interpreted such that if the pointer position is N% of the way down the scrollbar, the portion of the graph N% from the top is visible at the center of the window.

A.2.4 Graph Commands

The OPEN NODE command displays the code fragment corresponding to the selected node in a terminal window. Initially, the terminal window will be created as a bounding box and will have to be sized appropriately by holding down the middle button. IDeA allows the creation of five such edit/browse windows simultaneously. The name of the source file, the node name and node id will be displayed in the title bar of the edit window. The *vi* editor is invoked inside the terminal window. To close the window, the user must exit from the editor. If any changes have been made, the code fragment will have to be saved from within the editor. Until all the edit windows are closed, IDeA will not allow the user to read in a new graph or quit the current session. If edit windows are buried beneath other windows on the screen, the current window manager must be used to raise them to the top.

The OPEN EDGE command displays the true start and the true destination of a selected edge. The edge is selected by moving the cursor (small rectangular box) and clicking on the left mouse button. The edge information will be printed in the message window.

The COLLAPSE operation allows the user to collapse the subgraph rooted at a selected node into a subgraph node. Subgraph nodes and single entry nodes cannot be collapsed any further. If the collapse operation has to layout the graph again (due to external edges coming into subgraphs) the newly created subgraph node will be made to blink a few times to maintain context.

The EXPAND operation is the inverse of the COLLAPSE operation and expands the chosen subgraph. The nodes in the subgraph will be added to the currently displayed subgraph and the layout algorithm will be invoked.

The DESCEND operation allows the user to view the selected subgraph. The ASCEND operation displays the subgraph one level higher in the hierarchy.

The NEW ROOT operation displays the list of multiple roots at the highest level and prompts the user to choose a root through a dialogue box. This command allows the user to view graphs with multiple roots.

The REPLICATE operation allows the user to make multiple copies of a selected node. In the current implementation, only leaf nodes can be replicated. After a node has been selected for replication, the user will be prompted for a lower index and upper index for the replication.

BIBLIOGRAPHY

[ADA 68] Adams, Duane A., *A Model for Parallel Computations,* Parallel Processor Systems, Technologies and Applications, pp.311-333.

[AllK 82] Allen, J.R., and Kennedy, K., *PFC: A program to convert Fortran to parallel form,* Report MASC TR 82-6, Dept of Mathematical Sciences, Rice University (March 1982).

[AKPW 83] Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J., *A Conversion of Control Dependencies to Data Dependencies,* Proc. 10th Annual ACM Symp. on Princ. of Prog. Languages, pp.177-189, (January 1983).

[ABKP 85] Allen, R., Baumgartner., Kennedy, K., Porterfield, A., *PTOOL - A Semi-automatic Aid for Parallel Programming,* Draft - (November 1985).

[BRO 85] Browne, J.C., *Formulation and Programming of Parallel Computations: A Unified Approach,* pp.624-631, ICPP (1985).

[BRO 86] Browne, J.C., *Framework for Formulation and Analysis of Parallel Computation,* Parallel Computing 3, 1986, 1-9.

[BDN 86] Browne, J.C., Dutton, J.E., and Neuse, D.M., *Introduction to Graphical Programming of Simulation models,*

[DAV 82] Davis, Allen L., and Keller, Robert M., *Data Flow Program Graphs,* Computer, pp.26-41, (February 1982)

[DES 88] Deshpande, S.R., and Browne, J.C., *A Synthesis Approach to Algorithm-Specific Pipeline Design,* 1988 Architecture Conference (submitted).

[EAS 88] Easwar, S.R., *A relational database for the storage and manipulation of dependency graphs,* M.S. Thesis, Dept. of Elec. Engg., UT Austin, (May 1988).

[FOL 84] Foley, J.D., and Van Dam, A., *Fundamentals of Interactive Computer Graphics,* Addison-Wesley Publishing Company, (July 1984).

[KIM 87] Kim, S.J., *A General Approach to Mapping of Parallel Computations upon Multi-Processor Architectures,* Ph.D. Dissertation, Dept. of Computer Science, UT Austin, (Dec 1987).

[KUCK 77] Kuck, D.J., *A survey of Parallel Machine Organization and Programming,* Computing Surveys 9(1), (March 1977).

[KKLP 81] Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., *Compiler transformations of dependency graphs,* Eight ACM symposium on Principles of Programming Languages, (January 1981).

[LEE 87] Lee, T.J., *Software reuse in parallel programming environments,* Ph.D. Proposal, Dept. of Computer Science, UT Austin, (1987).

[McSHEA 86] Mary McShea., *Evaluation of Parallel Programming Languages,* TR-86-22, UT Austin, (September 1986).

[MEY 83] Carl Meyer., *A Browser for Directed Graphs,* M.S. Report, Dept. of Computer Science, UC Berkeley, (Dec 1983).

[PONG 86] Pong, M.C., *A Graphical Language for Concurrent Programming,* 1986 IEEE Computer Society Workshop on Visual Languages, (June 1986).

[ROWE 86] Rowe, Lawrence A., et al., *A Browser for Directed Graphs,* UCB Tech report UCB/CSD 86/292.

[SOB 86] Stephen Sobek., *Architecture Independent Parallel Programming,* Ph.D. Proposal, Dept. of Computer Science, UT Austin, (1986).

[SOB 88] Stephen Sobek, Mohammed Azam, and J.C. Browne., *Architectural and Language Independent Parallel Programming: A Feasibility Demonstration,* ICPP 1988 (submitted).

[SUG 84] Sugiyama, K., *A Readability Requirement in Drawing Digraphs: Level Assignment and Edge Removal for Reducing the Total Length of Lines,* Research Report No. 45, International Institute for Advanced Study of Social Information Science, Fujitsu Ltd, (March 1984).

[XLIB 86] Gettys, J., Newman, R., and Fera, T.D., *Xlib - C Language X Interface Protocol Version 10,* Massachusetts Institute of Technology, (November 1986).

VITA

Ramachandran Sriram was born in Madras, India, on September 13, 1963, the son of Poorani and C.S.Ramachandran. After completing his work at the Padma Seshadri Bala Bhavan School, Madras, in 1980, he entered the Regional Engineering College Tiruchi, India. He received the degree of Bachelor of Engineering in Electronics and Communication Engineering from the University of Madras in January, 1985. In August, 1985, he entered the Graduate School of The University of Texas.

Permanent address: 3709 Tom Green
Austin, Texas 78705

This thesis was typeset[1] with LaTeX by the author.

[1]LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin thesis format was written by Khe-Sing The.