Parallax: An Implementation of ELGDF
(Extended Large Grain Data Flow)

by
Inkyu Kim

A Research Project Submitted in Partial
Fulfillment of the Degree of Master of Science

Major Professor
Dr. Ted. G. Lewis

Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Professor Ted G. Lewis, for his help, guidance, understanding, support and encouragement. Also, I would like to thank Dr. Bella Bose , Dr. Walter Rudd my other committee members for their constructive criticisms and helpful suggestions.

I express my gratitude to the members of OACIS, Hesham El-rewini , Scott Handley, David Judge, who gave me constant encouragement and help.

I thank my friends S. Choi, H. Kim, for their help and encouragement.

Best thanks go to my parents, Hee-Tae Kim and Byung-Ok Choi, for their love, supports and encouragements throughout my life.

Finally, I thank my wife, Geeyeoun and my son, Hyunwoo for their patience and love.

Table of Contents.

List of Figures

Abstract

The major obstacle to widespread parallel programming of multiprocessors is the lack of a convenient parallel programming system[19]. PPSE (Parallel Programming Support Environment) is a unified approach to parallel programming. Parallax is developed as a component of PPSE based on ELGDF which is a graphical language for designing parallel programs. The goals of Parallax are to solve the following problems:

1) How to represent parallelism naturally in an application,
2) How to make a parallel program portable across different parallel computers,
3) How to remove time-dependent problems from the programmer's concern,
4) How to provide a standard software description format that can be used by other tools such as automated schedulers and performance analyzers, and
5) How to increase programmer's productivity.

Our approach to these problems are:

1) ELGDF notation: Parallax uses ELGDF notation that allows a wide representation of a variety of parallel programs in a natural way for both shared-memory and message-passing models using higher level parallel abstractions. Program details such as synchronization are handled by the system using reusable libraries specific to each target system. Parallax is a CASE tool which supports common Software-Engineering techniques such as hierarchical design concepts that support both top-down and bottom-up design using visual-programming techniques.

2) PP Design File: Parallel program designs are stored in a standard format in a PP (Parallel Program) Design File. This representation can be transformed into a task graph representation at different levels of granularity. The task graph of the problem can be used to estimate execution time and performance of the resulting parallel program.

Parallax is implemented on Macintosh as part of the PPSE project. It has been used to represent the design for both shared and distributed memory machines, with individual programs written in FORTRAN and C, with and without Linda support, and it has successfully produced program designs which can be analyzed by other tools such as TaskGrapher[5] and SuperGlue[6]. Parallax currently does not automatically produce a task graph, nor does it fully represent programs written in high level languages such as FORTRAN. Finally, ELGDF currently lacks formal semantics for representing computations.

1. Introduction

Increasing efforts are being carried out to develop parallel programming languages suitable for parallel computing [13]. But, the evolution of parallel programming languages has not kept up with the evolution of parallel architectures. The PPSE (Parallel Programming Support Environment) project is a unified approach towards parallel programming. The main purpose of this paper is to describe the design and implementation of Parallax: a forward engineering component of the PPSE project. This introduction section deals with the problems in parallel programming, a description of PPSE and explanation for developing PPSE and where Parallax fits in PPSE.

1.1 Motivation.

Writing parallel programs is significantly more difficult than writing sequential programs. Programmers of parallel programs must deal with a variety of new issues, including program partitioning , synchronization, concurrence, communication, fault tolerance, allocation and problems involving portability and compatibility between different parallel and sequential architectures. Even seemingly simple tasks(for example, detecting termination) become complicated and error-prone in parallel programming. These problems are inherent in the nature of parallelism and exist independently of the hardware [11]. For these reasons, we need to help parallel programmers.

1.2 Existing Approaches to Parallel Programming.

Followings are major existing approaches to parallel programming and critical assessment of each.

1.2.1 Enhanced versions of sequential programming languages.

Currently, most parallel programmers use enhanced versions of sequential languages to express parallelism. Enhanced version of sequential languages adapt some parallel constructs such as fork/join, parbegin/parend, doall, etc. to express parallelism. These enhanced versions are efficient for implementing parallel programs. However, these languages are machine dependent- each for a particular computation mode. Architectural details are exposed to the user. Program development is less productive since the designer is responsible for details of partitioning, synchronization, communication, and allocation etc..

1.2.2 Developing Intelligent Compilers.

With this approach, parallel algorithms are written in a conventional sequential language such as Pascal, FORTRAN, C. Intelligent compilers are developed to detect parallelism in sequential code and convert it into parallel machine code. This approach is user friendly because programmers can continue to use their familiar sequential languages. However, this approach is not flexible because parallel algorithms have to be sequentially coded. The efficiency depends on the intelligence of the compiler [13]. Most advanced compilers can enhance the performance by only a limited factor, due to the difficulty in exploiting global parallelism, which humans can do better. Also this approach is not always efficient, because the best algorithm for sequential code is not always best in parallel code.

1.3 What others are doing.

Some frontier research groups are trying to solve these problems. There are two major approach to new parallel programming problem : 1) Language Layering for Concurrency 2) New Language/Compiler [18].

1.3.1 Language Layering for Concurrency:

We can add a new language layer on the top of an existing language to describe the desired concurrency and necessary synchronization, while allowing the basic application programs to remain relatively unchanged. In enhanced versions of sequential programming languages, the most difficult to fix errors are made in synchronization-related code or in partitioning functions/data. These errors are concerned with parallel constructs not in the original programming languages. So it is good idea to develop a new language form to specify the parallel constructs and use an existing programming language to specify the detailed algorithm. Many groups are taking this approach.

Among them, CODE (Computation Oriented Display Environment) takes the most interesting approach [8]. They used visual technique to represent parallel constructs. In this section, a brief description of CODE is given.

CODE takes a unified approach to parallelism in trying to establish a basis for parallel programs that can be transported across parallel architectures. Two basic elements that express parallel program in CODE are:

computation-unit: A computation-unit is represented by a node on the screen and has a form, called node-specification form, to be filled by the user. The typical computation-unit is a function or a procedure that has been defined in a sequential, high-level language. Node-specification form incorporates the source code of a node. A computation-unit is composed of a functionality and firing rule. Functionality of computation-units is the transformation of its input-dependency set to its output-dependency set. Firing rules specify the states of the input dependencies that enable the unit to execute.

dependency relations: Dependency relations are represented by arcs on the screen. Dependencies are described in separate forms from firing rules. Dependency relations compose computation-units into parallel-computation structures.

One advantage of CODE is in enhancing the reusability of the parallel program by separating dependency and firing rule specifications from the functionality specification. Another idea is to use generalized dependency graphs to integrate design and specification with programming. This project shows that it is possible to write architecture independent parallel programs.

Problems

General problems with systems that adopt graphics in addition to text such as CODE, is loosing information in the translation of diagrams to text. Actually, CODE does not use diagram information to generate schedules while Parallax is able to produce process-processor scheduling information from a graphic description, CODE needs additional textual information to perform the same task. The constructs of CODE are not rich enough to express all possible parallelism. For instance, Loop constructs can not be expressed Graphically.

One big advantage of using well-defined language model such as C, Pascal, etc.. instead of developing new one is that we can use dusty deck serial code. But there is no discussion of reverse engineering in CODE documentation. For CODE system, reverse engineering will be very hard because of complicated firing rules for each process.

CODE does not address the data partitioning problem which is very important for scientific computation.

1.3.2 New Language/Compiler

Another possible approach is developing entirely new programming languages and compiler systems that integrate the concepts of concurrency and synchronization with all existing views of describing computational algorithms [18].This strategy is extremely expensive and potentially very labor intensive. Given the tremendous investment in existing application codes, this option must viewed as either a last resort or a means for developing ideas on how we could gradually evolve our existing systems into a structure more amenable to parallel processing.

1.4 PPSE project.

The Parallel Programming Support Environment(PPSE) project is an attempt to create a unified approach toward parallelism. PPSE research involves the following topics [5]:

1) how to partition an application into parallel parts,

2) how to map parallel parts onto multiple processors,

3) how to optimally schedule and run parallel parts,

4) how to reverse engineer existing serial source code,

4) how to measure and analyze performance,

5) how to distribute data over a multi-processor network,

6) how to coordinate design, code, debug and analyze performance.

Fig 1.1 shows how we organized these problems in the PPSE project. Reverse engineering involves retrofitting existing sequential programs onto parallel computers. Forward engineering deals with the task of writing a new parallel program from scratch. PPSE contains a set of software tools designed to help parallel programmers deal with reverse engineering and forward engineering aspects of parallel programs.

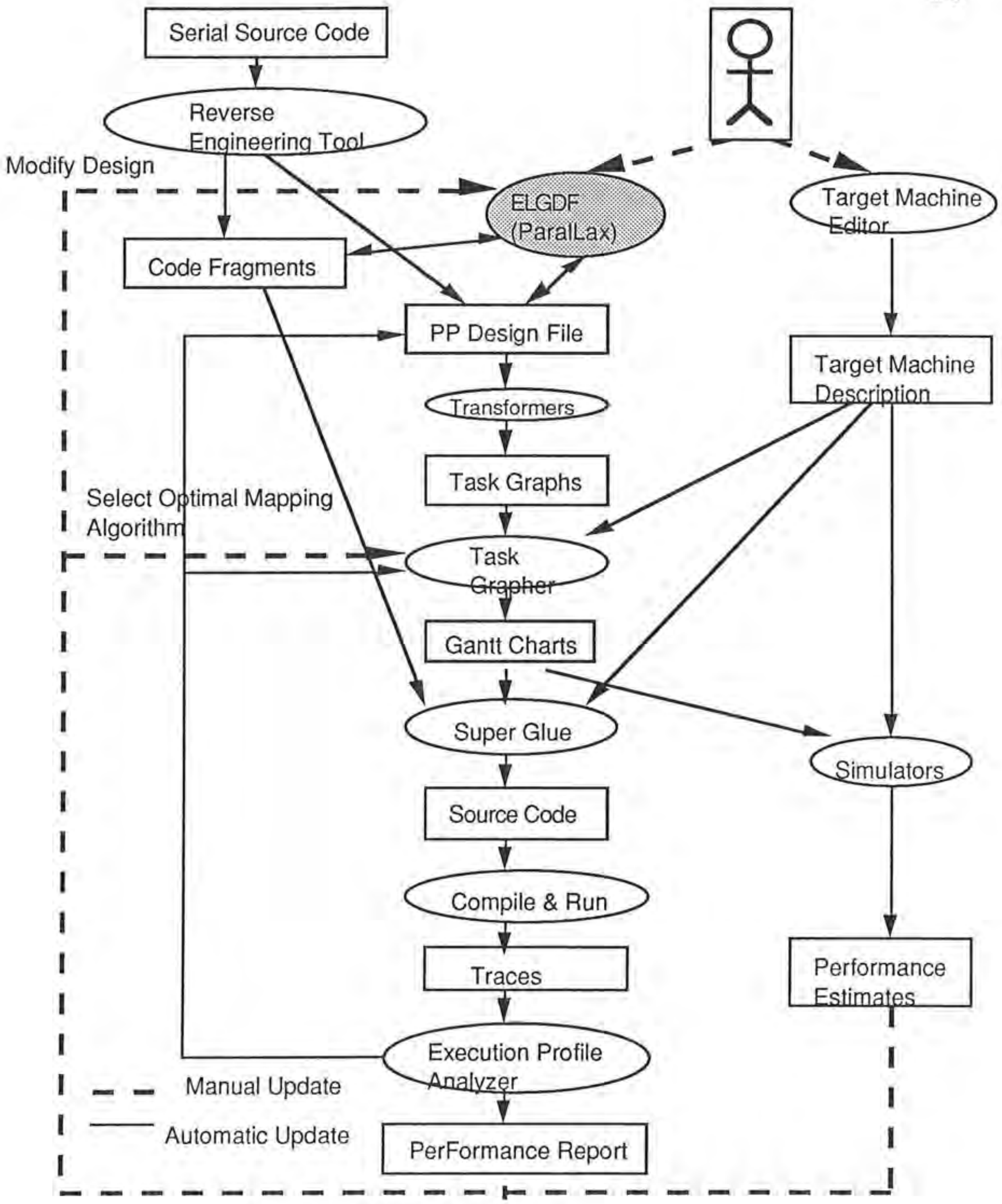


Fig. 1.1 PPSE Overview

1.4.1 Reverse Engineering

The reverse engineering part of the PPSE project involves converting serial source programs to parallel programs. Our reverse engineering tool analyzes an existing serial source code and generates Parallax compatible files. These files can be graphically displayed so that a user can modify and reorganize the program structure.

1.4.2 Parallax, a forward engineering component of PPSE.

The forward engineering part of PPSE involves how to design, implement, test, and evaluate the performance of a parallel program on specific hardware. Five tools have been developed. They are: Parallax, Target Machine Editor, TaskGrapher, SuperGlue, and EPA. Parallax is designed and implemented to work as one of forward engineering components of PPSE that allow a programmer to design and implement a parallel program in a natural way. Other three components of PPSE are briefly described in 1.4.3.

Design issues of Parallax.

1) How to write an architecture independent program.
2) How to represent a wide variety of parallel programs in most natural way.
3) How to provide information for mapping and/or scheduling.
4) How to reverse engineer existing serial code into Parallax form.

Approach

1) Higher degree of abstraction.

Raising the abstraction level allows program designers to express their algorithms in a higher level structure without having worry about the details of the specific hardware (Portability). Another advantage of a higher degree of abstraction is improving program maintainability. If the program can be broken into well-defined higher structured modules that are defined by their interface and what they do, not by how they do it. Changes in the programming of a module do not affect any of the rest of the program as long as the design discipline is adhered to.

2) Visual programming technique in addition to text.

Graphics has a growing role to play in programming. One of the purposes of using graphics in programming is to substitute some of text with some more natural, easy-to-understand means of expression. However, visualization of all program details overburdens the programmer and not efficient.

3) Provide a PP Design file that can be transformed into a task graph.

One nice feature of Parallax is providing information for scheduling tools and performance analyzer tools. The PP design file can be transformed into task graphs at different levels of granularity to be used by scheduling tools. Estimated execution time of tasks at different levels of granularity can also used by performance analyzer tools.

4) Support hierarchical design concept.

Parallax supports hierarchical design to allow construction and viewing of realistically sized applications. This also provides the flexibility for scheduling tools which means that can provide different task graph for different granularity.

Parallax provides a rich set of parallel constructs to express a variety of parallelism naturally, and adopts visual programming techniques for ease of use. Parallax takes full advantage of the information that is provided by the user in graphic description. For instance, we can generate Task Graphs from graphic descriptions for the use of scheduling and analyzer tools.

Parallax outputs Code Fragments and a PP Design File that is used by SuperGlue and Transformer (See Fig. 1.1). A detailed description of Parallax is given in Chapter 3.

1.4.3 Other components of PPSE.

Three other forward engineering components of PPSE, that work with Parallax, are briefly described in this section.

1) Target Machine Editor.

The Target Machine Editor is a graphical editor for giving a high-level description of architectures on which the parallel program might run. The Target Machine Editor provides the ability to graphically describe small irregular architectures or easily describe large regular architectures, graphically create shared memory, tightly coupled distributed memory or loosely coupled distributed memory architecture descriptions. Also some system specific information can be entered through the dialog boxes which are logically attached to the graphical icons. It produces a Target Machine Description File as an output to be used by TaskGrapher, SuperGlue, and Simulator [5].

2) TaskGrapher

The TaskGrapher is a tool which determines a schedule or map for assigning program segments to processors. It performs an automated mapping of the software onto the hardware, i.e. it maps program modules represented as nodes in a precedence task graph with communication (a transformation of the Parallax design file) onto arbitrary machine topologies and gives an allocation and ordering of tasks onto processors. It produces as output a Gantt chart, providing easy visualization of the allocation of the program modules onto the target machine processing elements, and the execution order of tasks allocated to each processing element. The Gantt chart consists of a list of all processing elements in the target machine. For each processing element, the Gantt chart shows a list of all tasks allocated to that processing element, ordered by execution time, including task start and finish times [6].

3) SuperGlue.

SuperGlue is a tool that automatically generates source code for a specific parallel computer from Parallax description of the software and the description of the target machine. SuperGlue is a software that takes a flow file (dataflow representation) of a parallel program (which is translated from the output of the Parallax), along with a target machine file, a Gannt file (which represents task scheduling) from TaskGrapher, and code fragments (from Parallax) and generates a parallelized version of source code that can be run on the described architecture [5].

4) Execution Profile Analyzer (EPA)

EPA is a tool that measures the performance of a parallel program. Timing routines are used to generate an execution profile of an application. These routines are the common interface of performance analysis to the SuperGlue source code generator. The timing routine generates a file which is called the "execution trace file". EPA uses this file as input and yields execution times of the grain and the communication delay or synchronization wait time as output which serves as input to TaskGrapher. It also produces a general report giving the overall speedup, overall and individual processor utilization and the time distribution among the grains which were mapped onto it.

2. Parallel programming

This chapter describes what components a parallel program should have and what steps are necessary to run the program on a parallel machine.

A parallel program must 1) define a set of sub-tasks to be executed in parallel, 2) specify when to start/stop their parallel execution, 3) specify coordination of parallel execution [15]. After writing a parallel program, this program must be scheduled to run on specific hardware.

2.1 Defining Parallel Subtasks

Defining subtasks refers to the decomposition of a large computation to a number of subprograms which can be executed concurrently. Defining subtasks of a program in a way that exploits as much parallelism as possible with lowest possible overhead is not easy. Parallelism and Scheduling with synchronization are the two most important factors influencing performance. Both of these factors depend on the granularity of code. If tasks are too fine grained, there is a time penalty for scheduling each node and synchronizing each arc which must be added to the execution time of the program. If the tasks are too large grained, which reduces the time penalty, all the available parallelism will not be exploited. Furthermore, when our target machine is distributed machine we must partition the tasks in a way that minimizes the communication between other tasks.

2.2 Starting and stopping parallel execution

A parallel program must specify when parallel tasks should start and stop execution. Current parallel programing languages provide abilities (fork/join, parbegin/parend, doall etc.) to express this.

2.3 Coordinating Parallel Execution

A parallel program must have a way to express the Coordination of Parallel Execution. Two mechanisms, Synchronization and communication can be used for this purposes. Synchronization involves the activation and suspension of concurrent processes to ensure correct results. Communication is the scheme for information exchange between processes. Synchronization is needed for communication and communication can be used for synchronization [11].

1. Communication- Shared Memory and Message-Passing

Cooperating tasks of a parallel program must communicate with each other. Two mechanisms that are currently popular, Message-passing and Shared-Memory. A parallel program must have functions to describe these two mechanisms.

2. Synchronizing Concurrency-Sequence Control, Access Control

Two synchronization mechanisms , Sequence control(control flow) and Access Control(mutual exclusion) can be used. Sequence control constrains events to make sure they happen in the right order and Access Control controls the access to a section of code in such a way, that only one competing process may enter at a time.

2.4 Scheduling program on given architecture.

Scheduling is the assignment of tasks to processors under time constraints. Scheduling can be either static or dynamic.

In static scheduling, tasks are allocated to processors during the algorithm design by the user or compiler. Scheduling costs for this method are paid only once even if the program is run many times with different data. Moreover, there is no run time overhead. The disadvantage of static scheduling, however, is possible inefficiencies in guessing the run time profile of each task.

Dynamic scheduling by the machine at run time offers better utilization of processors, but at the price of additional scheduling time.

2.5 Classification of Parallel Programming Languages.

A variety of programming models based on classes of architectures are described briefly in appendix A. The methods of exploiting parallelism are different in each class and are oriented toward utilizing specific aspects of the hardware. This section discusses how these parallel constructs can be represented in parallel programming languages. Parallel programming languages can be classified according to four attributes [19]:

1) Implicit or explicit parallelism?

Does the language have explicit constructs for concurrency(e.g. cobegin/coend, doall, tasks) or is the parallelism implicit(e.g. dependence analysis of "dusty decks")?

2) Implicit or explicit partition?

If the language has explicit parallelism, then is the partition implicit(e.g. doall) or explicit(e.g. tasks)? The partition of a parallel program specifies the sequential units of computation in the program and hence the granularity of execution.

3) Implicit or explicit schedule?

If the language has explicit parallelism and explicit partition, then is the schedule implicit or explicit(e.g. explicit use of processor numbers)? The schedule of a parallel program specifies the mapping of computations onto processors.

4) Shared-memory or distributed-memory?.

Is the communication model in the programming language based on a shared global memory? Any non-shared-memory model can be easily implemented on a shared-memory multiprocessor, but this issue is relevant because the converse is not true. In general, it is very difficult to efficiently implement a language with a shared memory model on a message-passing multiprocessor.

The next chapter describes how Parallax provides all necessary functions and information described in this chapter.

3. Parallax

This chapter describes Parallax. Some descriptions of Parallax syntax are also given. For more detailed information, refer to [2].

3.1 What is Parallax ?

Parallax is a design editor that provides a visual method of inputting software design details in ELGDF notation. Ideally, parallel software should be designed independent of any specific hardware on which the developed code might eventually run. Parallax allows the development of a high level, machine independent description of a parallel program. Parallax also allows the design of parallel software without being bound to any particular programming language. The following are major features of Parallax:

1) ability to produce a hierarchical design for parallel software in ELGDF notation,
2) ability to add information to graphic notation through dialog,
3) easy manipulation of design by re-sizing, encapsulating, and expanding the graphical description,
4) ability to add detailed textual specification to specific graphical notation of algorithm as code fragments,

3.2 Why Parallax ?

Parallax allows the development of a high level, machine independent description of a parallel program. Parallax also allows the design of parallel software without being bound to any particular programming language.

3.3 Functions of Parallax

Parallax describes parallelism and partitioning explicitly in the program. Each parallel construct is described in this section. The scheduling will be done implicitly by the scheduling tools.

3.3.1 Basic Constructs

1. Node: A circle, shown in Figure 3.1, can represent either a simple or a compound node. A simple node consists of sequentially executed code and is carried out by at most one processor. A compound node is a decomposable high level abstraction of a subnetwork of the program design network.

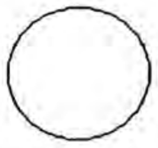


Fig 3.1 Node

2. Storage: A storage construct is represented by a rectangle, shown in Fig 3.2, and can represent either a storage cell(red color) or a collection of storage cells(blue color). A storage cell represents the data structure to be read or written by a simple node. A compound storage cell means constituents of the storage collection, but the details are given in a lower level description.

Fig 3.2 Storage

3. Split and Merge : Split and merge, shown in Figure 3.3 and 3.4 each, are special purpose simple nodes for representing conditional branching. Split has at most two output control-arcs; one for T = True, and the other for F = False. According to the truth or the falsehood of the condition associated with the split node one of its two output control arcs is activated.

Merge has N input control arcs and one output control arc. Two different kind of merges are defined according to its firing rules. OrMerge activates its output arc when it gets activated by any one of its N inputs. AndMerge activates its output arc when it gets activated by all of its N inputs.

Fig 3.3 Split Fig 3.4 Merge

4. Loop: A loop can represent For, While, or Repeat structures. Parallax represents serial loop compactly without using cycles in the graph. This is possible by describing only one loop body, and then specify a set of attributes such as the control variable, initial value, step, and loop bound in case of "For" or the termination condition in case of While or Repeat. A For loop iterated N times of the loop. Similarly, a While or Repeat structure can automatically be represented in terms of split, merge, node and While or Repeat constructs.

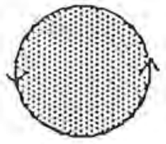


Fig 3.5 Loop

5. Replicator: A replicator, as shown in Figure 3.6, is one of the parameterized constructs in Parallax that represents concurrent loop iterations compactly. A set of attributes is associated with the replicator such as the control variable, initial value, step, and replicator bound. Replication of a node N times produces N concurrent instances of that node. An arc connected to a replicator is expanded as a set of identical arcs each of which is connected to one of the replicated instances.

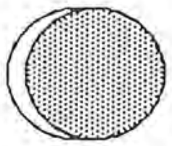
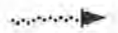


Fig 3.6 Replicator

6. Arcs: An arc in Parallax can express either data dependency, sequencing, transfer of control, or read and/or write access to a storage construct. A set of attributes is associated with each arc to provide information about the arc type, data to be passed through the arc, storage access policy, and communication strategy. An arc can be either a simple arc(red color) which cannot be decomposed or a compound arc which is decomposable into a set of other simple and/or compound arcs(blue color).



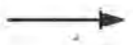


Fig. 3.7 Control Arc   Fig 3.8 Data Arc

Simple arcs can be classified into control and data arcs. A control arc, as shown in Fig. 3.8 (dotted line) expresses sequencing or transfer of control among nodes. A data arc, shown in Fig. 3.9, carries data from one node to another or can connect a node to a storage construct. A data arc connecting a node and a storage construct can represent READ, WRITE, or READ/WRITE access according to the direction of the arc. A data arc can be used to carry data once or repeated n times per activation. One of the arc's attributes is used to indicate the number of times the data will be passed through. If the value of that attribute is greater than one then the arc is considered a repeated arc. The repeated arc is used basically in pipelines. It can carry data (repeated times) from a simple node to another in a synchronized fashion. Also it can express synchronized writing and reading to or from a storage cell.

3.2.2 Convenient Structures.

Parallax also supports many of the common structures in parallel programs that can be synthesized using the constructs. It automatically provides them for program designer convenience.

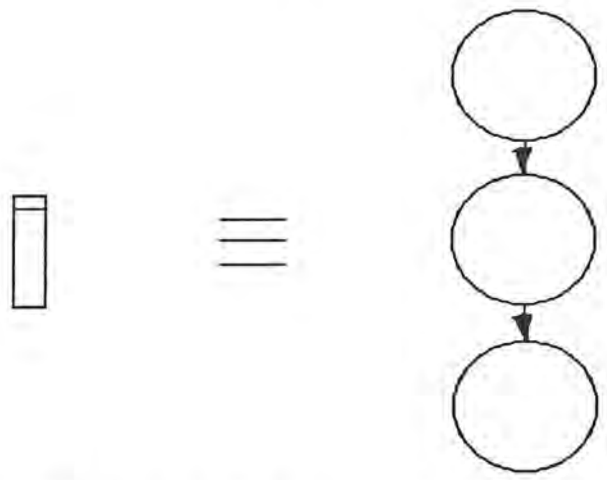


Fig 3.9 Pipe and its equivalent

A pipe, as in Fig 3.7, is a high level abstraction that represents a set of N nodes forming a pipeline . The pipe consists of N simple nodes and N-1 m-repeated arcs. The nodes forming the pipeline are replications of the same simple node. A pipe has several attributes associated with it such as number of stages in the pipeline (N), number of times the data will be passed through repeated arcs in the pipe (m) and others.

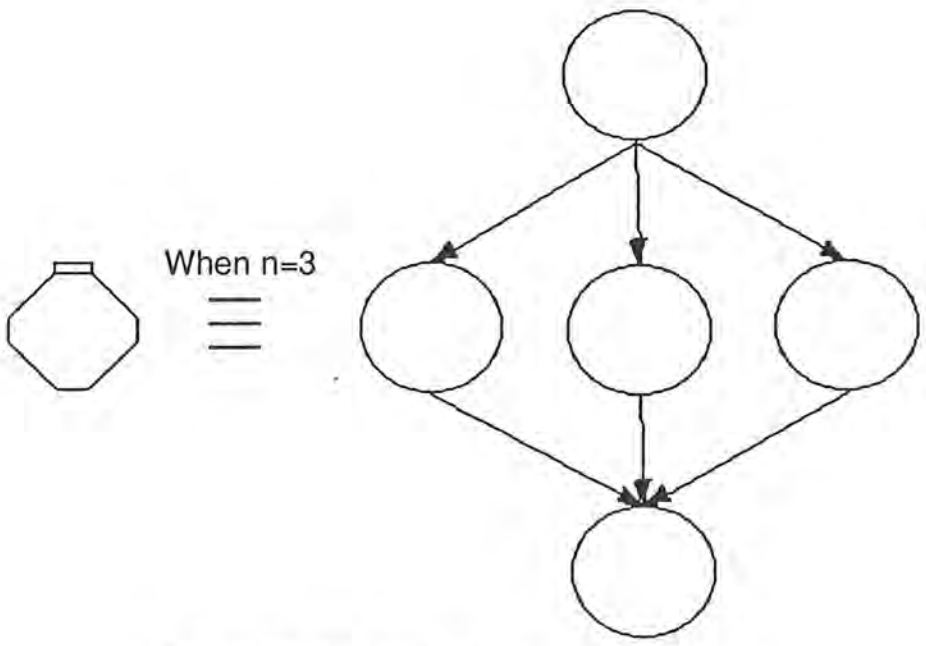


Fig. 3.10 Fan and its equivalent

A fan of size n is composed of a start node S, n parallel nodes, 2n control arcs, and an end node (E) as shown in Fig. 3.11. The start node activates the parallel nodes and when they all finish E gets activated. The size of fan should be constant. Compound arcs that are connected to a fan carry data to or from its constituents.

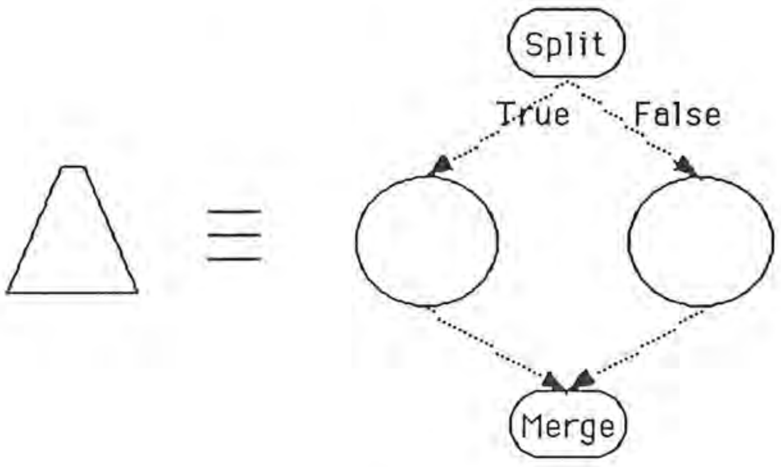


Fig. 3.11 IfThenElse and its equivalent

IfThenElse and fans are examples of common structures. The system can prepare skeletons for the types of structures per designer request. Using these structures reduces the drawing time, helps design readability and comprehension, gives more information for analysis tools.

3.3.3 Mutual Exclusion

Parallax helps designers to easily express mutual exclusive access to shared variables by having an attribute associated with each arc connecting a node to a storage construct. If the exclusion attribute is set, then mutual exclusion is guaranteed. In Fig. 3.10, A and B can access X in any order. Both A and B want to update X through a Read/Write arc and that might produce an incorrect result unless we set the mutual exclusion attribute associated with those Read/Write arcs to guarantee mutual exclusive access to X.

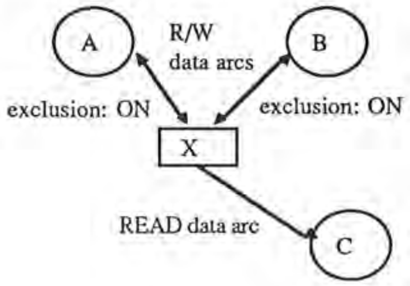


Fig. 3.12 Access Control in Parallax

3.3.4 Hierarchical Design

Parallax supports hierarchical design to allow construction and viewing of realistically sized applications. In Fig.3.11, a compound node NodeA is a higher abstraction of node a0, a1 and arc da1. A compound Arc d2 is higher abstraction of d21 and d22. Only bottom level node has some source code associated with it. This feature make integrating design and coding phase of software developing cycle possible.

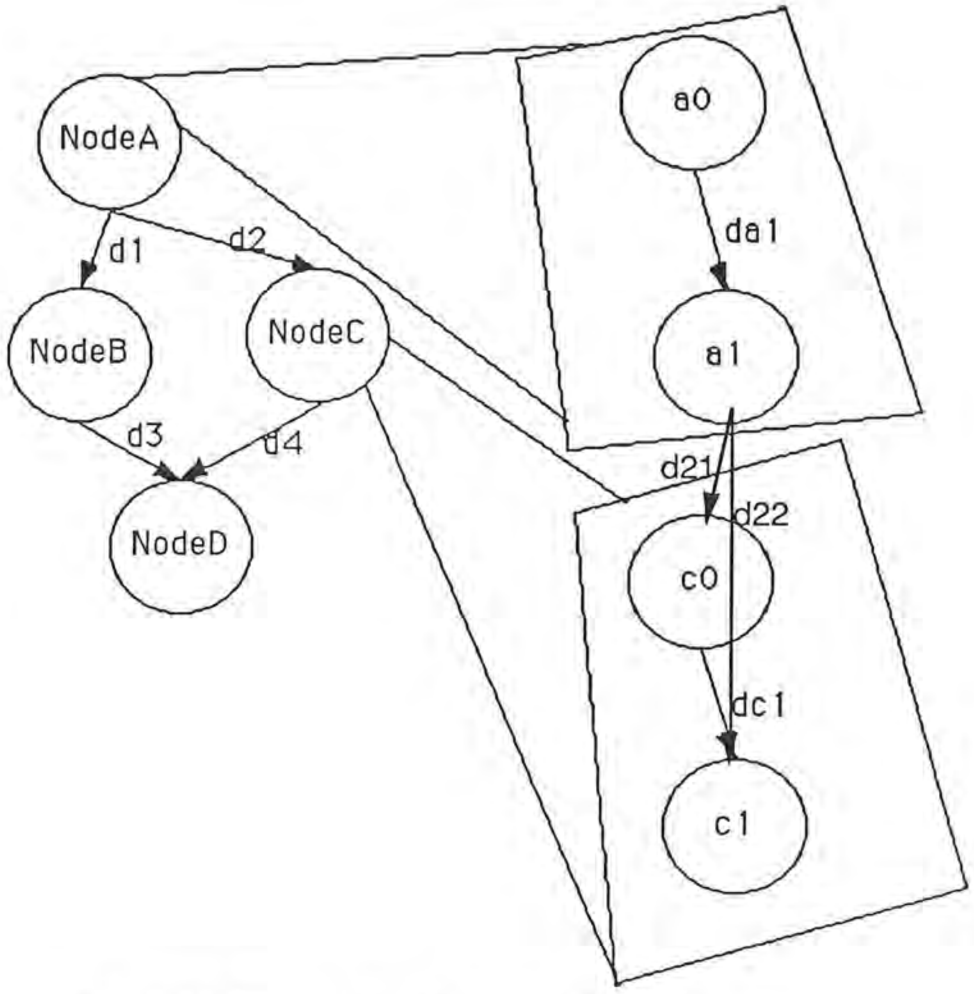


Fig. 3.13 Hierarchical Design in Parallax

4. Using Parallax

This chapter shows how to design and write a parallel program with Parallax . An example will make things clear. This chapter also shows how use tools and menus.

4.1 Menus

This section describes each of the Parallax menu commands. It is organized by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear in the menu.

4.1.1 Apple Menu

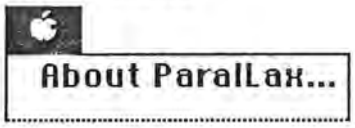


Fig 4.1 Apple Menu

This command tells you what version of Parallax you are using.

4.1.2 File Menu

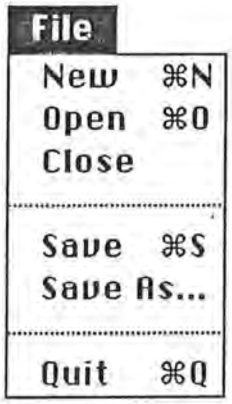


Fig. 4.2 File Menu

New

This command opens a new PP Design window. This window is top level window, and can not be closed until all other windows are closed. This command also open a tool window(Palette). Tool window will not be closed until top level window is closed.

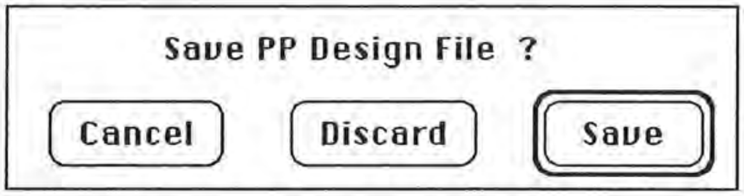
Open

This command lets you open an existing PP Design file on top level window. It also set up tool window(Palette).

Close

Basically this command close active window. Following are some rules for closing window:

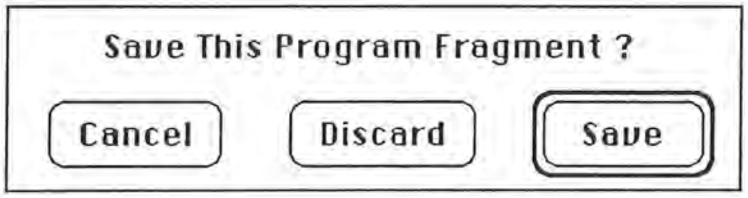
If current active window is top level window, then close current PP Design if there is no other opened window. If current PP Design file is not saved, system will ask you if you want to save current PP Design file.


Save PP Design File ?

Cancel Discard Save

Fig. 4.3 Dialog for saving PP Design file

If current active window is a text window, system will ask you if you want to save current text file(program fragment) if it is modified and not saved.


Save This Program Fragment ?

Cancel Discard Save

Fig. 4.4 Dialog for saving Code Fragment

If current active window is a documentation window, system will ask you if you want to save current documentation file if it is modified and not saved.

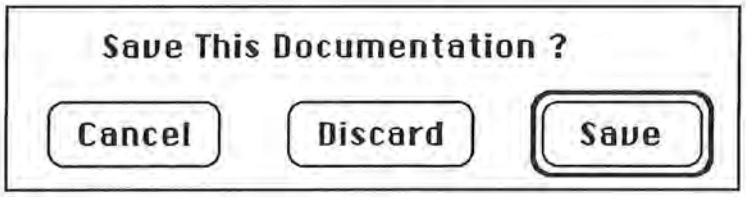


Fig. 4.5 Dialog for saving Documentation

If current active window is graphic window but not top level, system will just close the window.

Save

This command saves the file in the active window if it is text window. If the current active window is top level window, system saves current PP Design file. If the current active window is graphic window but not top level, system does nothing.

Save As

This command lets you the current file under other name. All others are same as Save.

Quit

This command quits Parallax and returns to the Finder. If project has been modified and not saved, system shows the dialog showed at Fig. 4.3

4.1.3 Edit Menu

This menu will be enabled if current active window is either text or documentation window. The Edit menu has the standard Macintosh editing command.

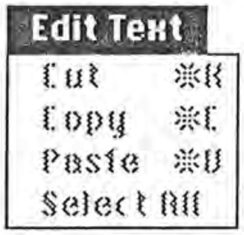


Fig. 4.6 Edit Text menu

4.1.4 Tool Menu

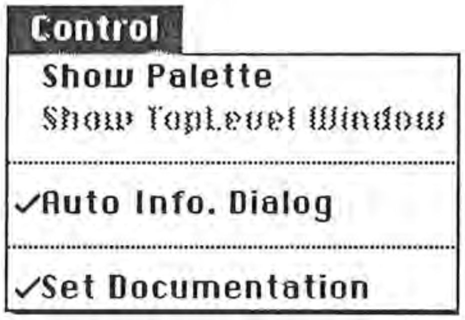


Fig. 4.7 Control menu

Show Palette
This command shows the palette if it is hidden by some other window.

Show TopLevel Window
This command makes top level window front window if front window is not top level window.

Auto Info. Dialog

When a user draws a symbol or an arc with checking Auto Info. Dialog, the system automatically shows a dialog so that user can enter information for the symbol or the arc right after he/she creates the symbol.

Set Documentation

Checking Set Documentation has two different features:

When double click on a symbol or an arc with select tool, the system checks Set Documentation is selected. If selected, it opens a documentation file otherwise open a dialog for the symbol or the arc.

Set documentation is also used to enable or disable documentation window. If Set documentation is selected, user can edit the contents of the documentation window otherwise its just for browsing the documentation.

If the documentation is not in the current directory, the system show a dialog to ask user if he want to find or making a new documentation file for that symbol.

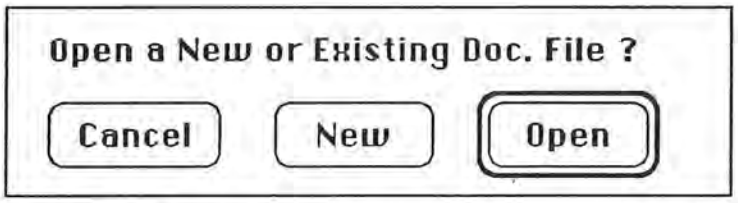


Fig. 4.8 Dialog for opening documentation file

4.2 Tools

This section describes the tool palette. The palette has two different kinds of tools: 1) manipulation tools, 2) drawing tools.

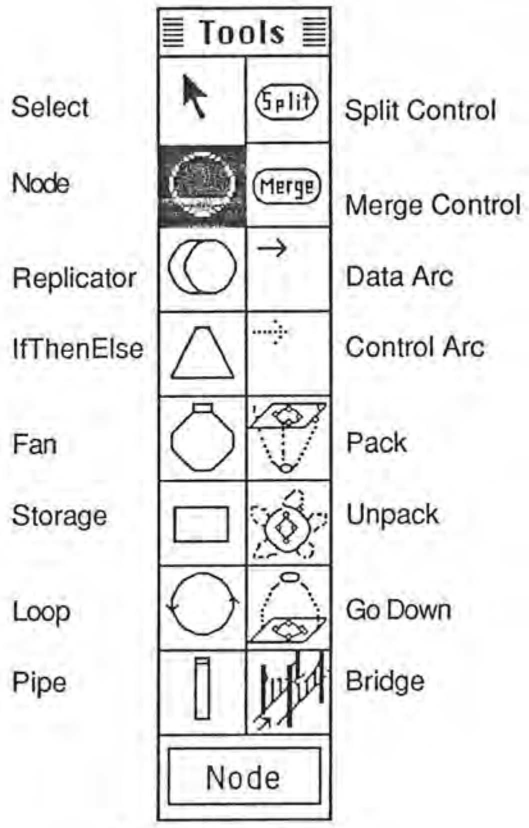


Fig. 4.9 Palette

4.2.1 Manipulation tools

Manipulation tools are for modifying and reorganizing the program. Four tools are dedicated for this purpose and their usages are described in this section.

Select Tool

This tool has four different functions:

1) To delete, first select symbol(s) you want to delete and hit delete button.
2) To move, symbol(s) pick a symbol and drag it to the place you want.
3) To enter information, double click on the symbol for you want to enter information. Appendix C shows all different dialogs for each different symbol and describes how to enter information.
4) To show documentation file, double click the symbol after selecting Set Documentation from control menu. The system open the documentation file and shows a documentation window as described above(see Set Documentation).

If you change a symbol from compound to simple even though lower layers exist, system also will ask you if you really want to discard all layers below that symbol.

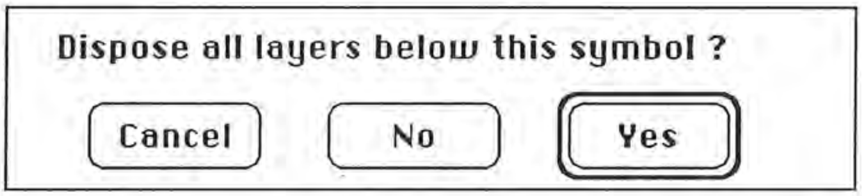

Dispose all layers below this symbol ?

Fig. 4.10 Dialog for disposing layers below the symbol

Pack Tool

For bottom-up design, the Pack tool allows user to compose symbol(s) as a node that represent higher level abstraction of the symbol(s). To do this, select symbol(s) with this tool, then the system will ask you if you want to pack.

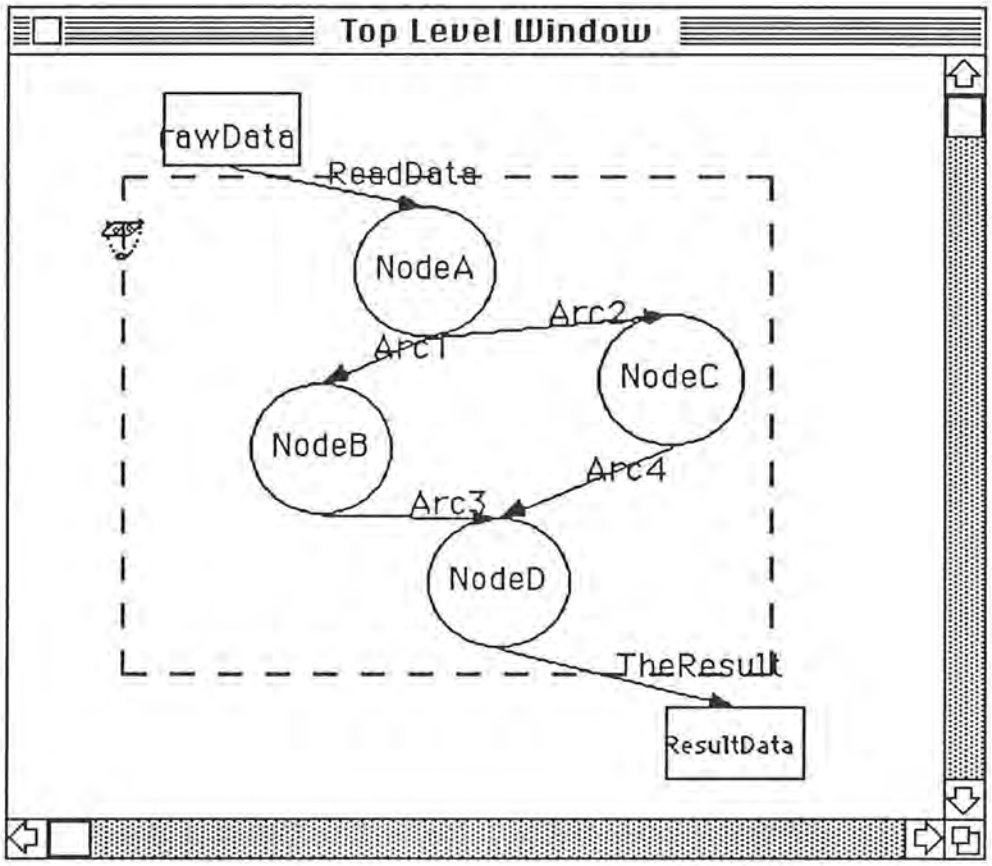
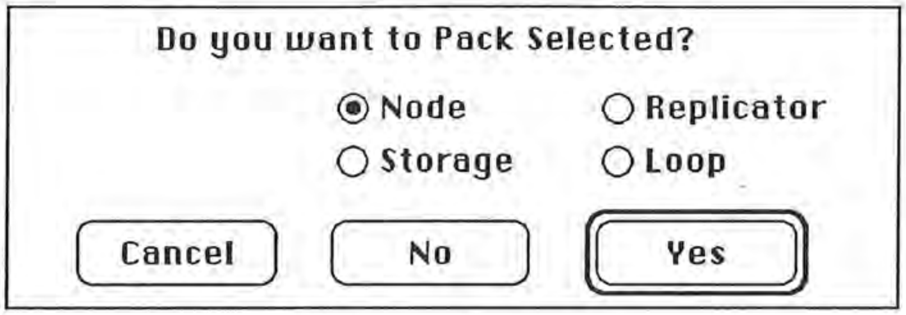


Fig. 4.11 Packing symbol(s).

Note that there are radio buttons that the user can select the symbol he wants for the packing.



Do you want to Pack Selected?

[✓ Node] Replicator Storage Loop

Fig. 4.12 Dialog for packing

Unpack Tool

Unpack tool allows the user to decompose a symbol if it is compound and has at least one next level symbol. To do this, double click on a compound symbol with this tool that has at least one symbol at a lower hierarchical level, then the system will display the next level of the symbol on current window if it is not opened. If there is no next level symbol, system will give you an alert. If a window is opened for that symbol, the system gives alert. If the user unpacks a Loop or Replicator, the system warns the user that he/she will lose the functionality of the Loop or Replicator. This tool is also needed if user make mistakes in Packing symbols. Fig. 4.14 shows the window after the user unpacks the symbol.

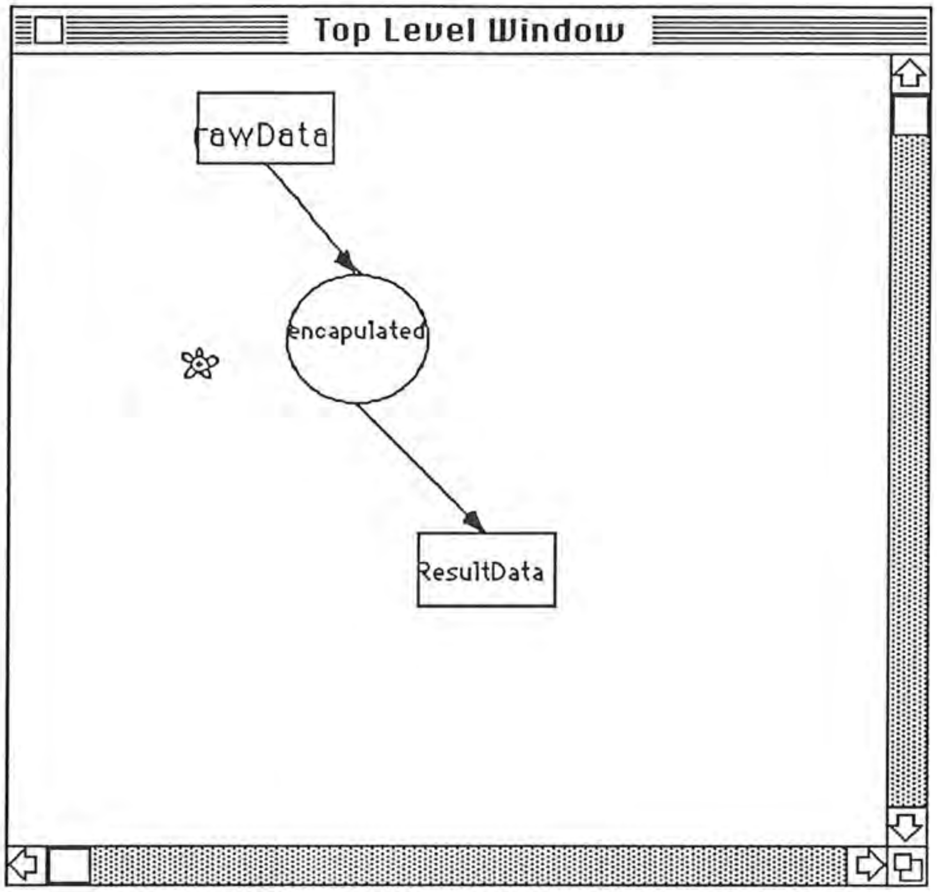


Fig. 4.13 Unpacking a symbol

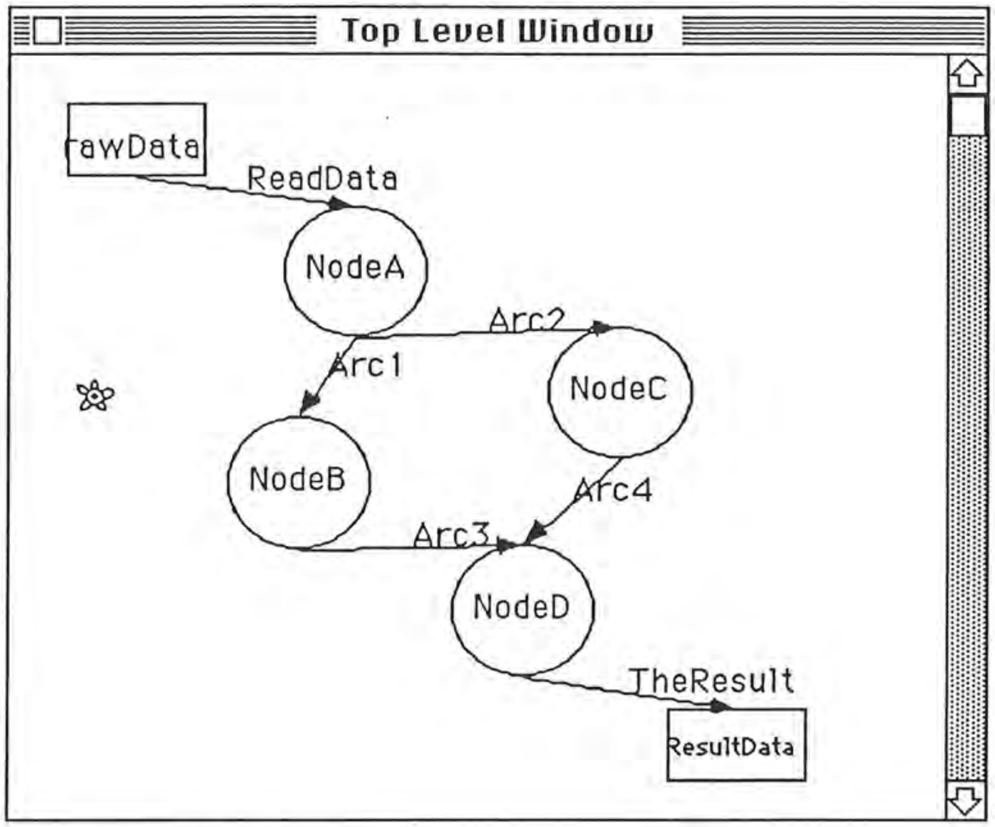


Fig. 4.14 After unpacking a symbol

GoDown Tool

If the user double clicks on a compound symbol which is an encapsulated node(as in Fig. 4.13) with this tool, the system opens a window and shows next level of that symbol (shown in Fig.4.15) if not opened already. If that window is already opened, system will make that window the front window so that user can specify next level of the symbol. This tool supports a top-down design concept.

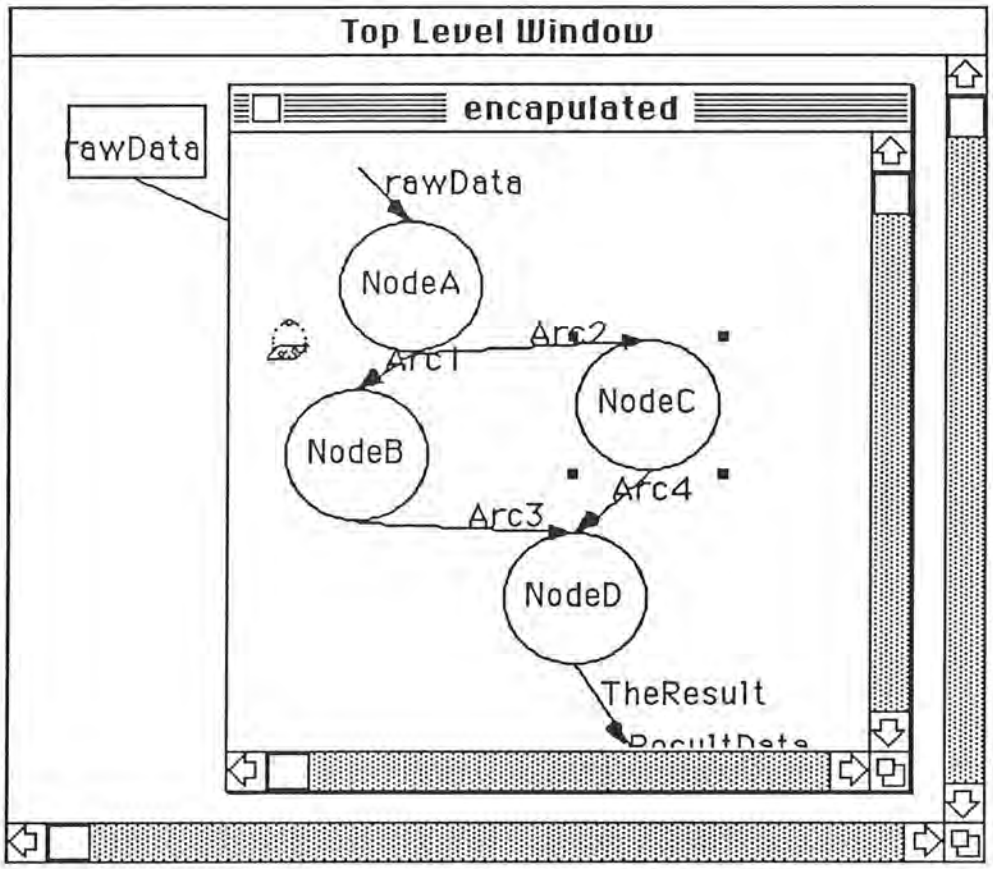


Fig. 4.15 Go Down

If you you repeat the above procedure on a simple symbol, the system will ask if you want to open a new text file or open an existing text file.

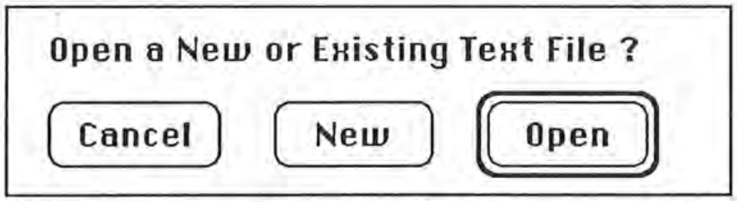

Open a New or Existing Text File ?

Fig. 4.16 Dialog for opening a text file

If you select **New**, the system opens a new text window so that you can enter code fragments for that symbol. If you select **Open**, System will show you standard dialog (shown in Fig. 4.17) so that you can select the file you want to open.

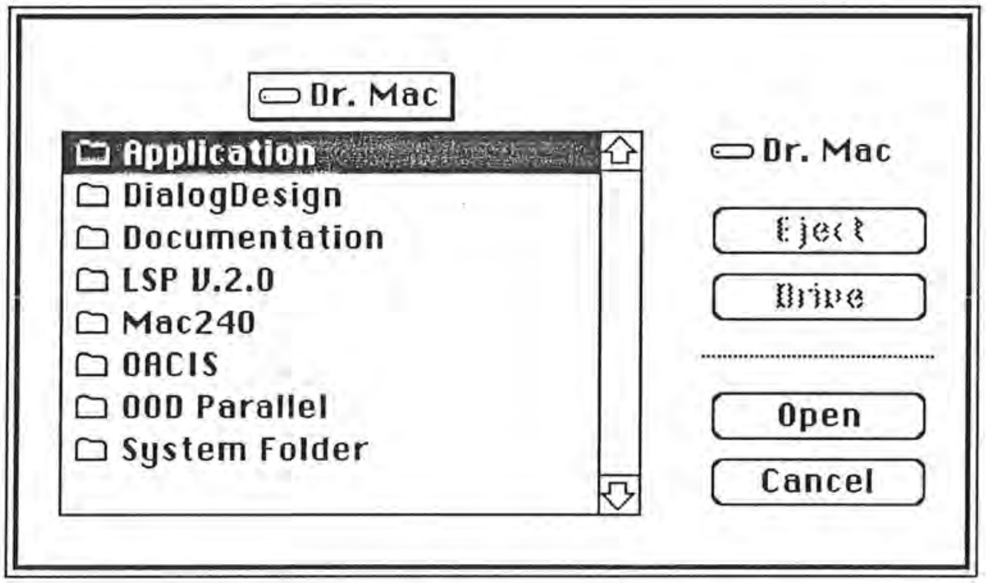


Fig. 4.17 Standard dialog for opening files

4.2.2 Drawing Tools

This section describes how to draw symbols and arcs, which represent computation units and communication, respectively, and how to enter information for them. There are two different kinds of drawing tools: One is for drawing symbols (symbol drawing tools include: **Node, Replicator, IfThenElse, Fan, Storage, Loop, Pipe, Split, Merge**) and the other is for drawing arcs between two symbols (arc drawing tools include: **Data Arc, Control Arc**). In addition, there is a special tool **Bridge** to connect two symbols from different window.

1) How to draw symbols and arcs on screen:

To draw symbols, select the symbol drawing tool you want. The cursor changes to a representation of the symbol. To make the symbol appear in the drawing window, click at the location in the window where you want the symbol to be located. To draw an arc between two symbols, select one of arc drawing tools and drag from one symbol to another. There are 9 symbol drawing tools and 2 arc drawing tools.

2) How to enter information for the symbol:

There are two ways to enter information for a symbol or an arc. First, the user can enter information for the symbol or the arc right after he/she draw by checking Auto Info. Dialog as explained before. Secondly, if you double click on a symbol with a symbol drawing tool or select tool, the system will show a dialog for the double clicked symbol so that user can enter information for the symbol. To enter information for an arc, select the select tool and double click on an arc. The system will show you a dialog so that you can enter information for the arc. Appendix C shows all information dialogs for symbols and arcs and description of each session for each dialog.

Bridge

This tool lets you connect a symbol from one window to one from another window. To connect two lower level symbols, their upper level counterparts should be connected first. At the lower level, the user must choose a source and destination symbol. These should agree with the flow direction at the higher level. The system will check for flow direction consistency. In the source dialog, cancel and no does the same thing (for complying Macintosh user interface convention).

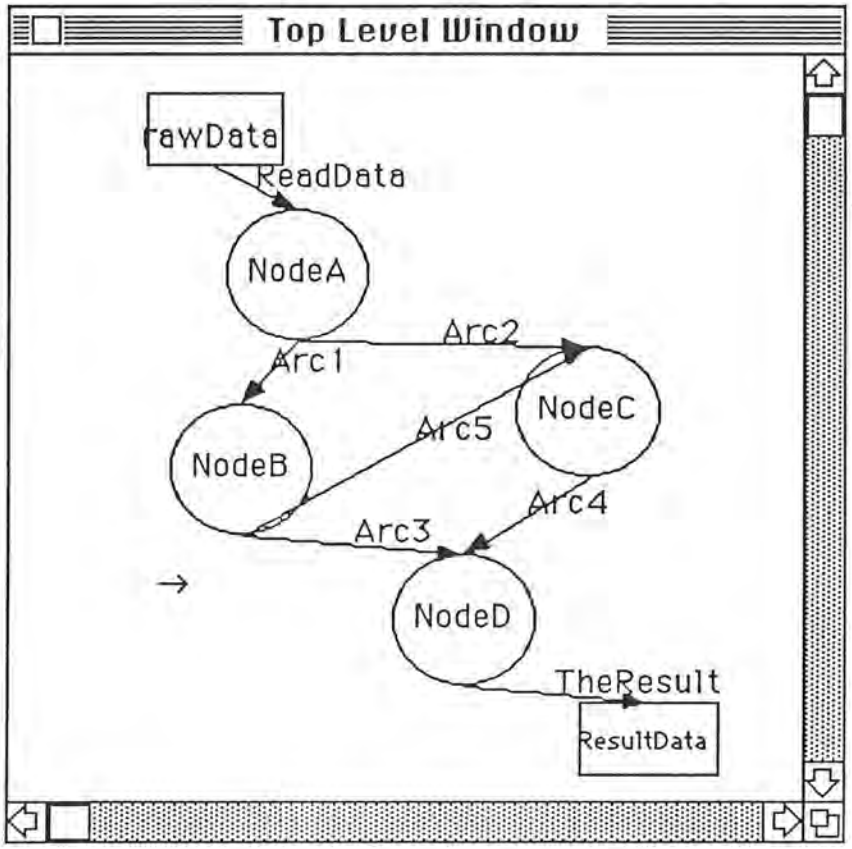


Fig. 4.18 Connect upper level first

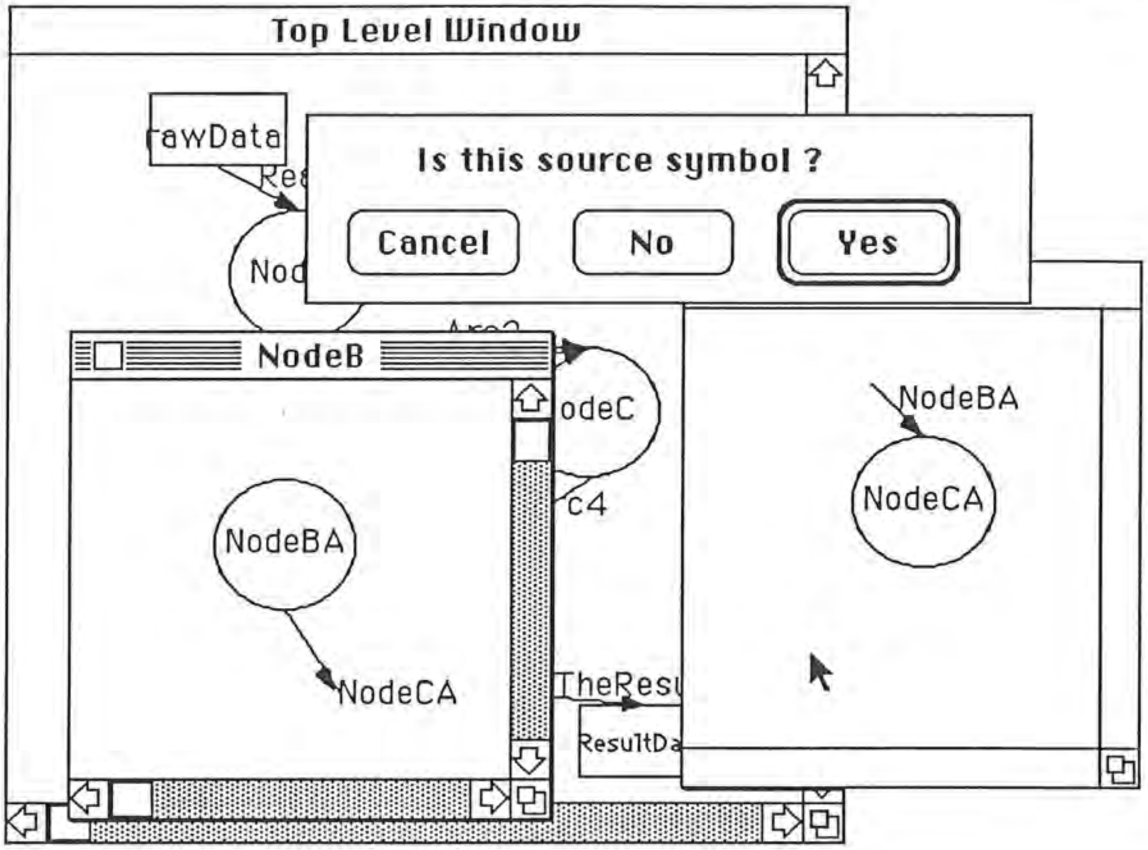


Fig. 4. 19 Dialog for selecting source symbol

When you select destination symbol, the system shows a dialog so that you can specify the type of arc. Note that this dialog has radio buttons to specify whether the arc is data arc or control arc. Cancel button in this dialog means canceling source symbol- the user must select a new source symbol. The No button means that this symbol is not the destination.

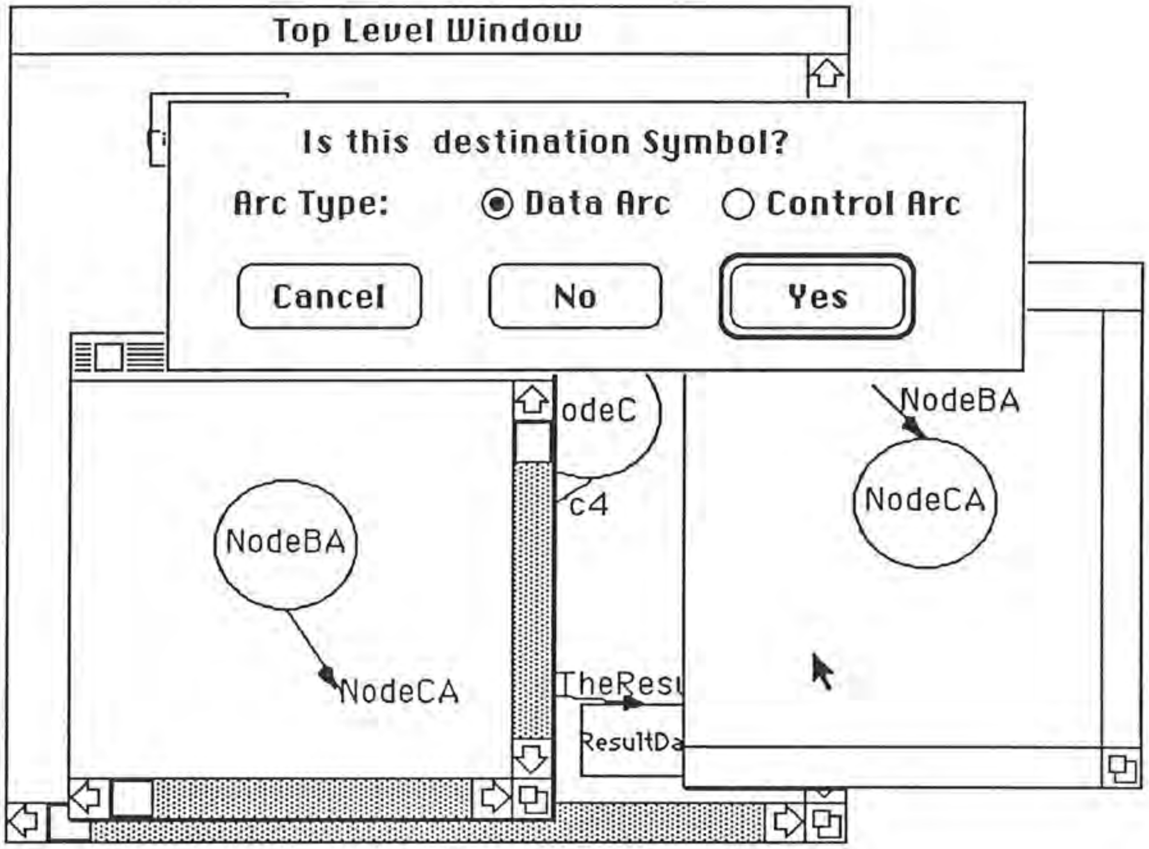


Fig. 4. 20 Dialog for selecting destination symbol

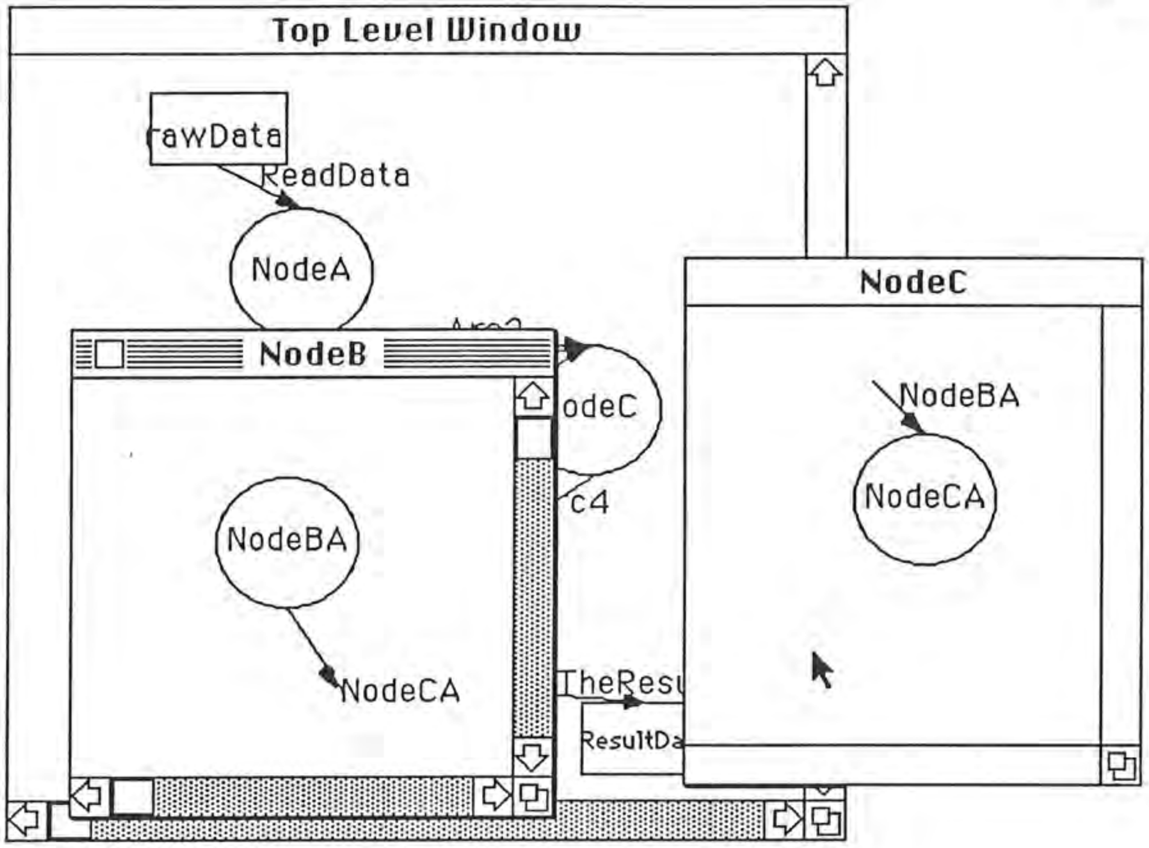


Fig. 4.21 After bridge

4.3 An example program: The Traveling salesman problem

The Traveling salesman problem is presented to show how to design a parallel program with Parallax. In this discussion we shall, without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge <1,k> for k <- V - {1} and a path from vertex k to vertex 1. We define g(1, V - {1}) is the length of an optimal salesman tour, when V includes all cities. From the principle of optimality it follows that:

$$g(1, V-\{1\}) = \min_{2 \leq k \leq n} \{c1k + g(k, V - \{1,k\})\}$$

Let's consider a simple example and show how to write a parallel program for this problem with Parallax. Consider the directed graph of Fig a. The edge lengths are given by the matrix Fig 4.22.b.

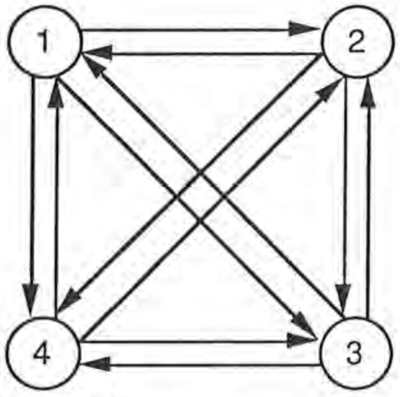


Fig.4.22.a map

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Fig. 4.22.b distances matrix

The definition is given in recursive form, but this can be converted to a FOR loop. A dynamic programming version of TSP is presented.

First step, refer from the distances matrix.

g(2,ø)= c21 =5; g(3,ø) = c31 = 6; g(4,ø) = c41 = 8;

Second step, calculate when the loop variable i is2:

g(2, {3}) = c23+ g(3,ø) = 15; g(2, {4}) = 18

g(3, {2}) = 18; g(3, {4}) = 20

g(4, {2}) = 13; g(4, {3}) = 15

Third step, calculate when loop variable i is 3:
(that means g(i,S) with |S| = 2 and i <> 1 and 1<-/ S and i<-/ S)

g(2,{3,4}) = min{ c23 + g(3,{4}), c24 + g(4, {3})} = 25;

g(3,{2,4}) = min{ c32 + g(2,{4}), c34 + g(4, {2})} = 25;

g(4,{2,3}) = min{ c42 + g(2,{3}), c43 + g(3, {2})} = 23;

Finally, calculate when loop control variable i is 4:

g(1,{2,3,4} = min{c12 + g(2, {3,4}), c13 + g(3, {2,4}),
c14 + g(4, {2,3})} = 35;

The Top level description of TSP in Parallax might look as in Fig 4.23.

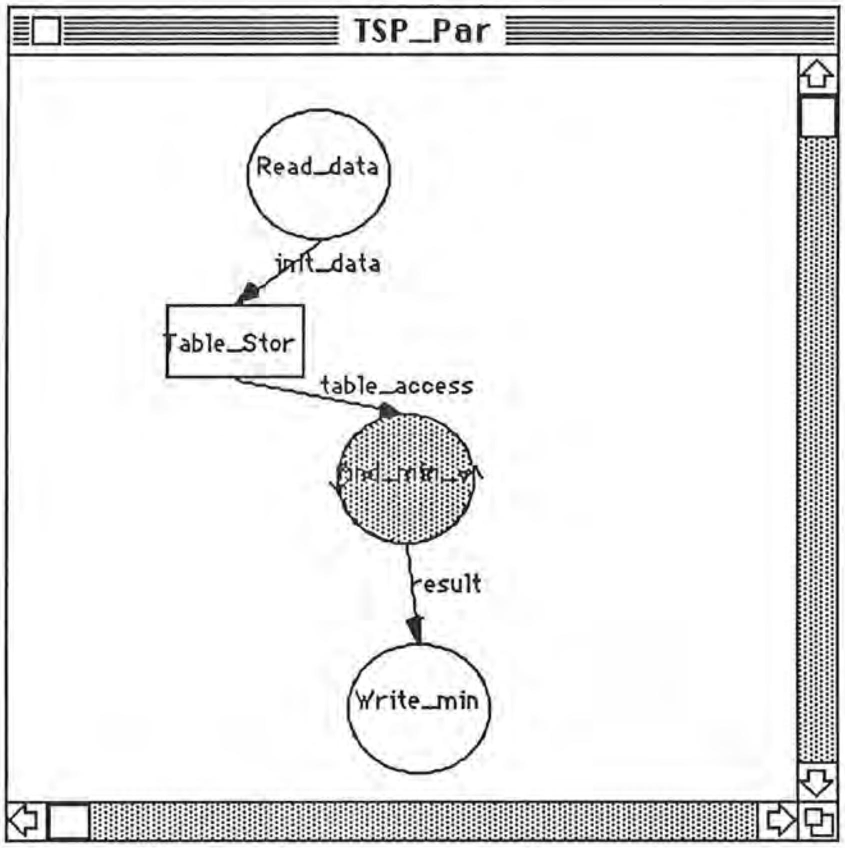


Fig.4.23 Top level design

ReadData node reads data like the number of cities and edge length matrix and put them in Table_Stor. The find_min_v is a compound loop. This loop repeats n-1 times and sends the result to the Write_min node. The Write_min node outputs the min value. Fig.24 shows the expansion of compound loop find_min_v.

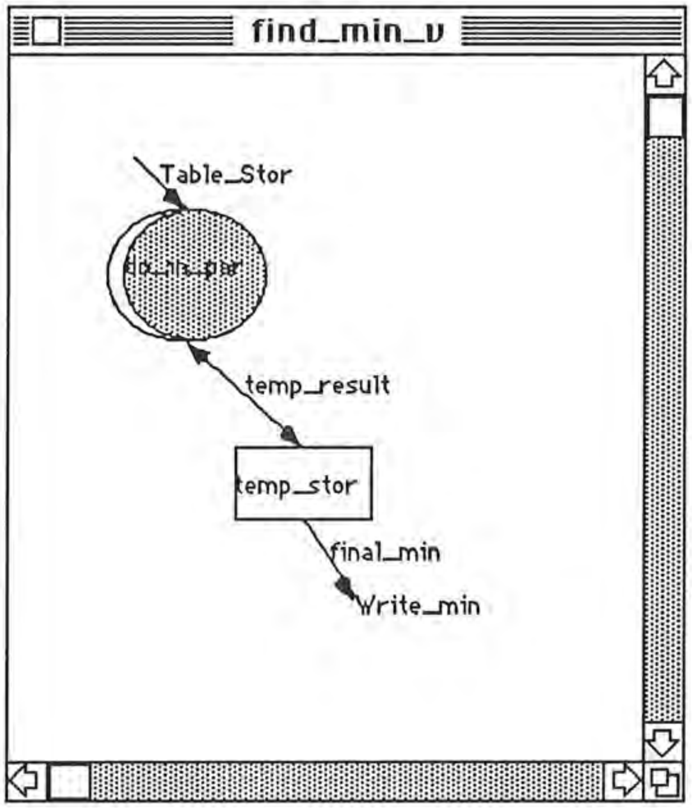


Fig.4.24 Expansion of find_min_v

This compound loop will be executed n-1 times from i = 2 to i = n. In find_min_v, $_{n-1}P_i$ processors will be executed in parallel. For example, when n = 4 and i= 2, 6 processors will execute in parallel, each computing one of 6 paths described in the second step of the example. Then do_in_par saves the result in temp_stor.

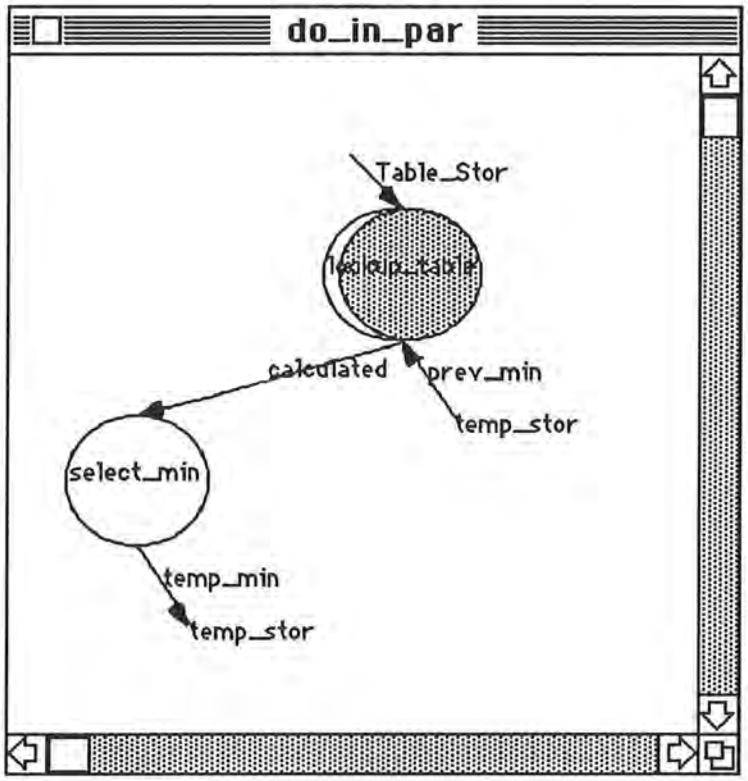


Fig.4.25 Expansion of do_in_par

The lookup_table replicator refers to two tables, one is Table_stor and the other is temp_stor to find the next step solution. Table_Stor has the original distances matrix and temp_stor has the previous results. The select_min node selects minimum value among the same starting and ending point paths. For example, when i = 3, it selects 25 for g(2,{3,4}).

Finally, when the compound loop finishes its execution, it sends the minimum value to Write_min. Write_min writes the minimum value to storage.

5. Implementation

5.1 Design Consideration

Making an Easy-to-use, fast, small, maintainable and portable software is not easy, because these attributes conflict with one another. Therefore, a designer must make "trade-offs", which is not easy either. The major design considerations in designing Parallax are :

- Easy to use:

Parallax takes advantage of icons and Macintosh Standard User interface. Parallax incorporates a menu bar, a group of icons that are on the Tools palette, basic working space.

- Portable

Portability is a big issue especially when many people work together. PPSE project has a lot of subprojects , some of them use different machines and moreover target machine may vary. For this reason, we chose ASCII File format so that we can communicate with each other.

- Maintainable

Structural design and OOD(Object-Oriented Design) concepts are rules that guided the design and implementation of Parallax. OOD takes more time to define modules and design inter-module dependencies, but the result source code is more understandable and modifiable.

5.2 Data Structure

To represent all the symbols and arcs on the screen, Parallax uses a clustered linked list in which symbols and arcs may be inserted, deleted, dragged or inspected at any moment. The linked list structure of Parallax is shown in Fig 5.1. Handles were used instead of pointers because handles are relocatable and will prevent memory fragmentation. An object has 6 handles, each of them used for different purposes. ObjectFriend Handle is used to point to the next object in the window. If there is an Arc between Object A and B then, these two objects are connected by SuccConn and PreConn, otherwise SuccConn and PreConn are set to NIL.SuccConn and PreConn are also handles, that form a linked list. ChildObject points to the first object of next level that associates with the object. WindowTo field points to the window that shows all objects of the next level. For example, WindowTo field of Object A shows all children objects of A. One handle, the OBJ_topHeadHdl points to the very first object in the list so that searching whole linked list is easy. Also windowObjectHdl points to the first object in the window for the same reason.

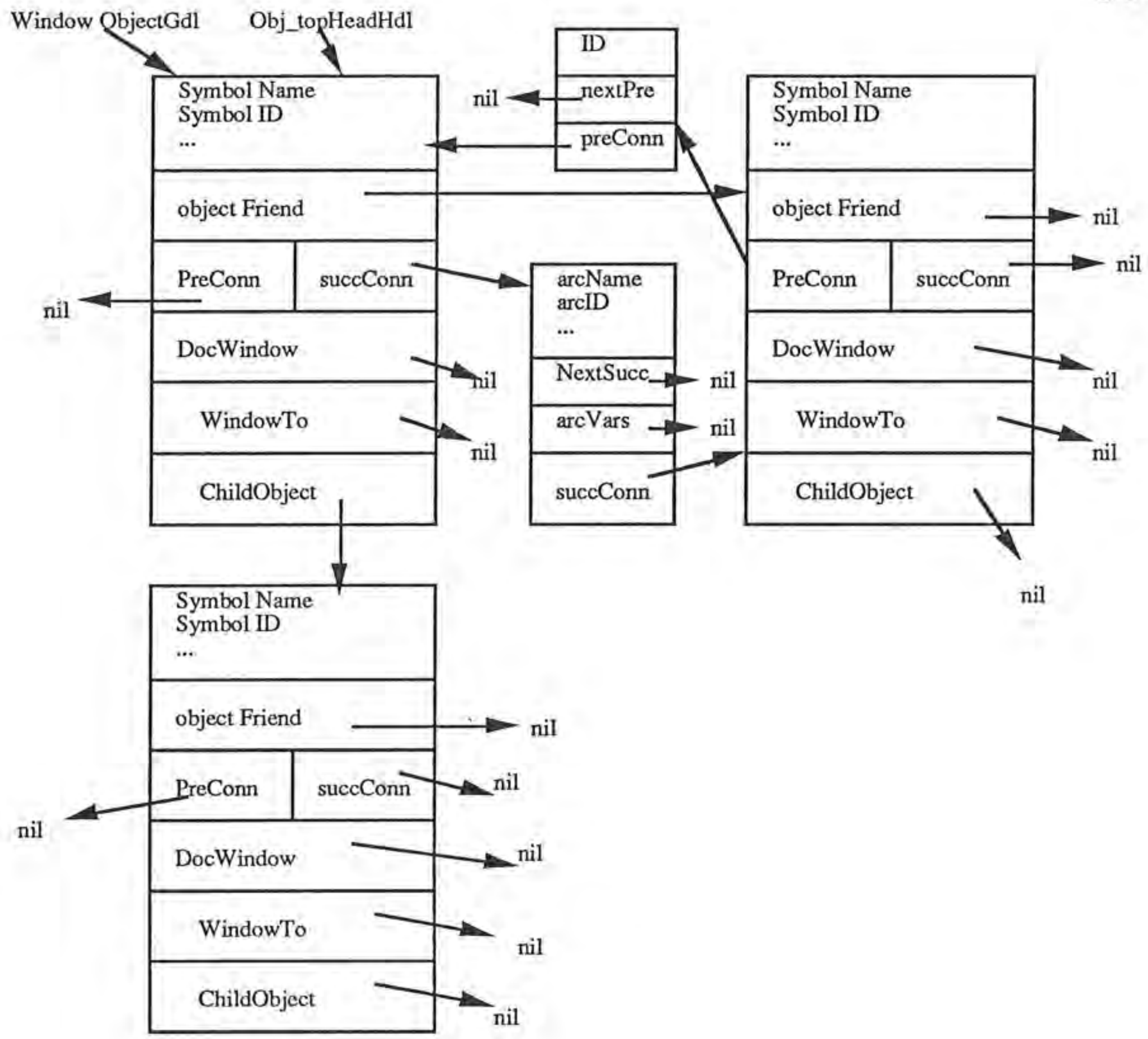


Fig 5.1 Data Structure

As an example, Fig 5.2 has 4 objects on the toplevel. NodeA has Arc to NodeB, NodeC,and has children Object a0, a1. Fig 5.3 shows data structure for Fig 5.2. example of PPDesign .

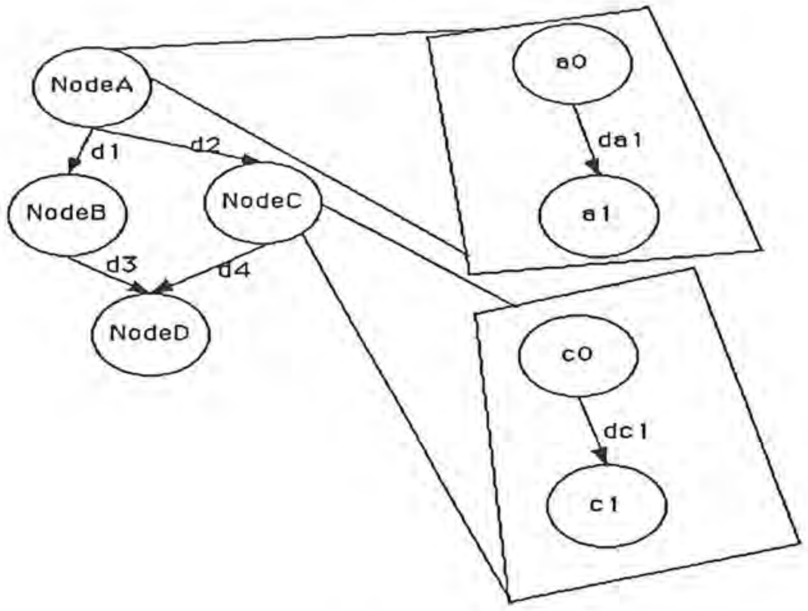


Fig. 5.2 An example of Parallax design

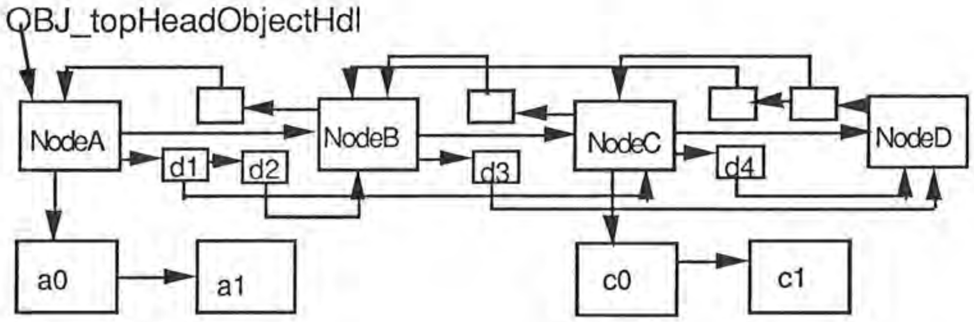


Fig.5.3 Top level data structure of Fig.5.2

6. Conclusion

Parallax is powerful enough to express parallelism and to provide information for automated schedulers and performance analyzers. Parallax also integrates the design and coding phase of the software life cycle. However, only through real practice and experimentation will the ultimate solutions to the parallel software design problem emerge.

My experience of Parallax is easy to use, a natural way to express parallelism and very useful tool for designing and implementing parallel programs.

7. Further Research

Parallax has several problems:
1) The syntax of the graphic notation is sometimes not clear.
2) According to the current definition of Parallax, a primitive symbol can have any type source code, but I think that a primitive symbol should have only procedure or function level code, otherwise the system provides heading part of it to make it procedure or function level. A good reason of doing this is to improve reusability of both source fragments and PP Design files.
3) Data arc definition in Parallax can represent almost anything. This is confusing. There are design inconsistencies in drawing arcs too.
4) Firing rules are also not clear. These unclear definitions make clear design difficult.

I think we need to test Parallax with more example programs to develop and redefine the tool to make it real good stuff.

Possible research areas are:

1) Developing a conversion algorithm that converts PP Design into task graphs.
2) Developing intelligent debugging tools.
3) Extending Parallax
   - How to minimize communication overhead among tasks automatically or semi-automatically.
   - How to improve reusability of both code fragments and PP Design file.
   - Refining the firing rule of each task.
   - Refining the rule for drawing arcs between tasks.
   - Type declaration functions.
4) Depveloping standard synchronization patterns
   Nonrepeatable errors(time dependant error) are most difficult to debug. In [18], it was pointed out that, "As long as programmers are responsible for managing synchronization, it will be extremely difficult to know that all possible timing errors have been eliminated". We need to develop standard synchronization patterns if possible and encapsulate them as completely as possible.
5) Develop a source-level tool for prediticting execution Times of tasks.
6) Providing user-defined structures.
7) Developing a formal representation of Parallax
8) Developing forms to express data parallelism.
9) Developing forms to express dynamic data structure.
10) Developing non-Linda version Code generator.
11) Developing libraries
12) Developing standard program partitioning patterns

Appendix A. Parallel Architectures

It is possible to discern four major divisions which have been widely examined: Pipeline and vector processors, SIMD machines, Shared memory architectures and Message passing multiprocessors [12]. Last two categories are our main concern in PPSE project.

A.1 Pipeline and Vector Processors

Pipeline and Vector Processors focus mainly on internal parallelism, i.e. parallel operation within a single processor. The architectures are characterized by high computation speed, large main memory and fast, large secondary storage support. Concurrency is achieved through multiple functions and logic units, assorted pipelines, multiple large high-speed register files, associative memory and fast memory access via interleaving etc. All of the architecture provide hardware support for vector instructions which operate on full vector operands through pipelining.

A.2 SIMD Machine

SIMD machine consist of a collection of synchronized processing elements(PEs) with an associated control processor. The control processor broadcasts the same instruction stream to all PEs. Each PE has some form of enabling facility such that if it is enable then broadcast instructions are executed on local data; this provides the ability to perform branching. This machines are useful for limited class of problems which have simple control flow and operate on large amounts of data. The main reason for this restriction is that the architectures are particularly susceptible to sequential code segments and branching; these result in poor processor utilization and load balancing difficulties.

A.3 Shared Memory Architectures

Theoretically, shared memory computers employs a large number of identical processors which share a common memory. The processors may read simultaneously in a single cycle; any number of processors may write to memory simultaneously. A memory cell to which a number of writes occur simultaneously contains one of the values written(or a random value/minimal value, etc.). For this reason, shared memory machines provide the ability to synchronize processes when many processors access a shared variable.

A.4 Message Passing Models

Message passing models have a collection of processors which do not share memory but communicate and synchronized by sending messages. Each processor is an independent MIMD machine with attached local memory. Two approaches to utilizing machines of this class have been investigated based on synchronous and asynchronous message passing.

Appendix B. Parallax File Format.

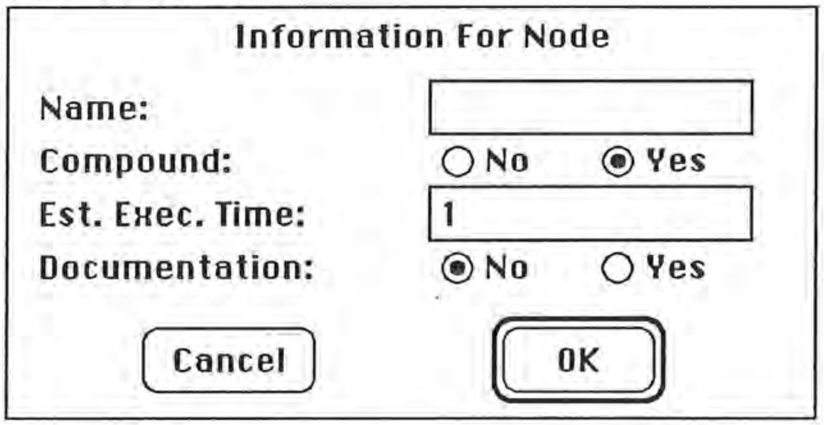
```
Parallax      -> Win
Win           ->'$**WinSt**$'     Symbol_List   '$**Win Ed**$'
Symbol_List   -> Symbol |Symbol Symbol_List |e
Symbol              ->'$Symb St$'  Symbol_Info   Pre_List  Arc_Data    '$Symb Ed$'
-> '$Symb St$'  Symbol_Info  Pre_List  Arc_Data  '$Symb Ed$' Win

Arc_data      -> '$StArc$'    Arc_List   '$EndArc$'
Arc_List      ->  e | Arc_Info | Arc_Info ArcList
Arc_Info      -> '()'  Arc_Rec    '()'

Pre_List      ->'(Pred Obj St)'  Preds   '(Pred Obj Ed)'
Preds         -> e | Pred  |   Pred Preds
```

Appendix C. Dialogs for symbols and arcs.

Followings are dialogs for different symbols and arcs. Brief description of each session for each dialog are given.

C.1 Dialogs for entering symbol information.

1) Node dialog


Information For Node

Name:

Compound: No [✓ Yes]

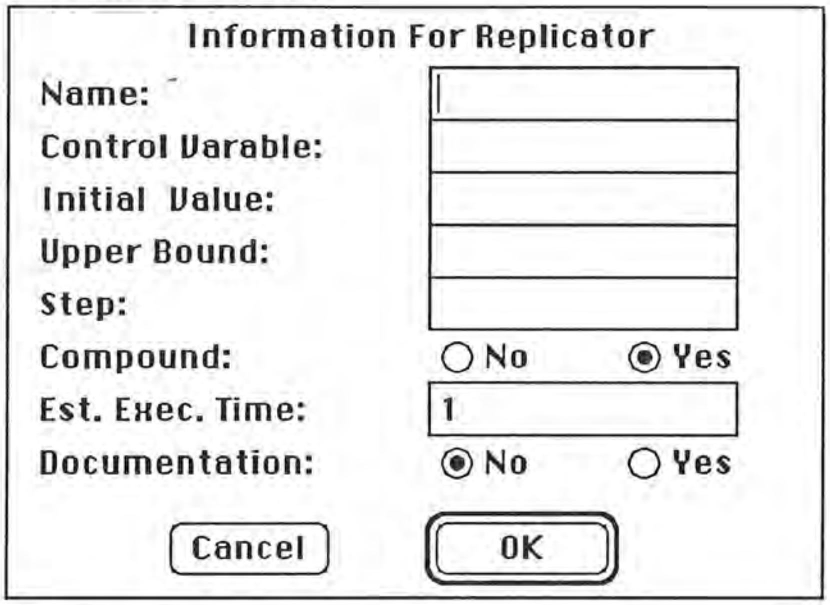
Est. Exec. Time: 1

Documentation: [✓ No] Yes

Fig. C.1 Dialog for NODE

Name: This editing session lets you specify name of the symbol.
Compound: These two radio button lets you specify whether symbol is compound or simple.
Est. Exec. Time: This editing session lets you enter estimated execution time of this node.
Documentation: These two radio button lets you specify whether documentation is ON or OFF. Initially, documentation is OFF and after you close this dialog, you won't see documentation. But if you set to documentation ON, you will see documentation window for that symbol after you close the dialog.

2) Replicator dialog



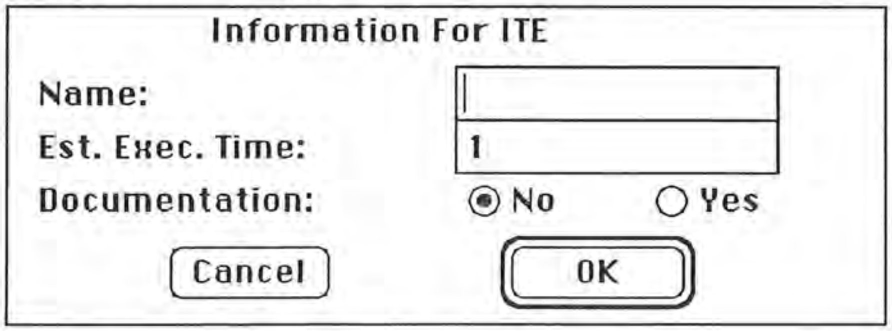
Information For Replicator

Name:	
Control Varable:	
Initial Value:	
Upper Bound:	
Step:	
Compound:	No [✓ Yes]
Est. Exec. Time:	1
Documentation:	[✓ No] Yes

Fig. C.2 Dialog For Replicator

Control variable: This session is for entering name of control variable.
Initial Value: This session is for entering initial value of the control variable.
Upper Bound: This session is for entering upper bound value of the control variable.
Step: This session is for entering step value of the control variable.

3) IfThenElse dialog

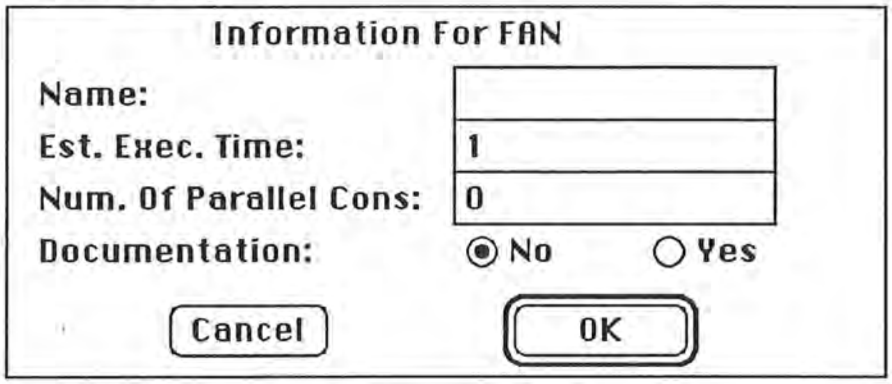


Information For ITE

Name:	
Est. Exec. Time:	1
Documentation:	[✓ No] Yes

Fig. C.3 Dialog for IfThenElse

4) FAN dialog



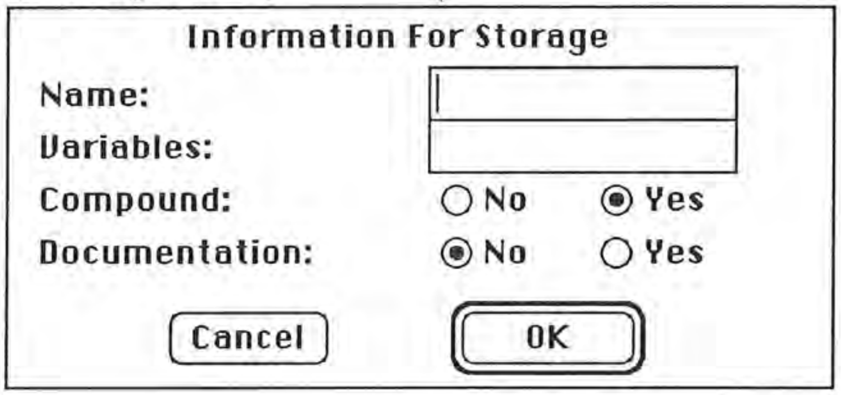
Information For FAN

Name:	
Est. Exec. Time:	1
Num. Of Parallel Cons:	0
Documentation:	[✓ No] Yes

Fig. C.4 Dialog for FAN

Number Of Parallel constructs: This editing session lets you specify how many parallel constructs this Fan has.

5) Storage dialog



Information For Storage

Name:	
Variables:	
Compound:	No [✓ Yes]
Documentation:	[✓ No] Yes

Fig. C.5 Dialog for Storage

Variables: This editing session lets you specify name of variables in this storage. If you type variables, the system will show dialog shown in Fig. C.6 for each variable. Variables are separated by "," if more than one.

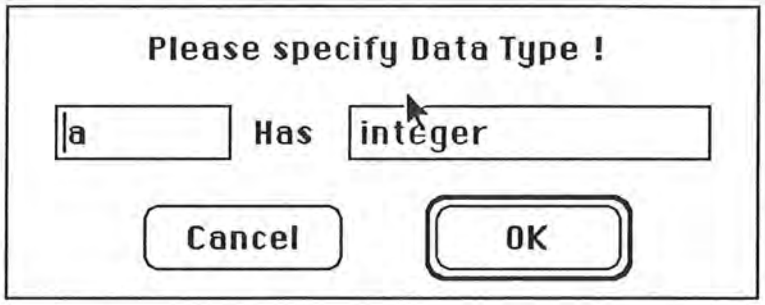
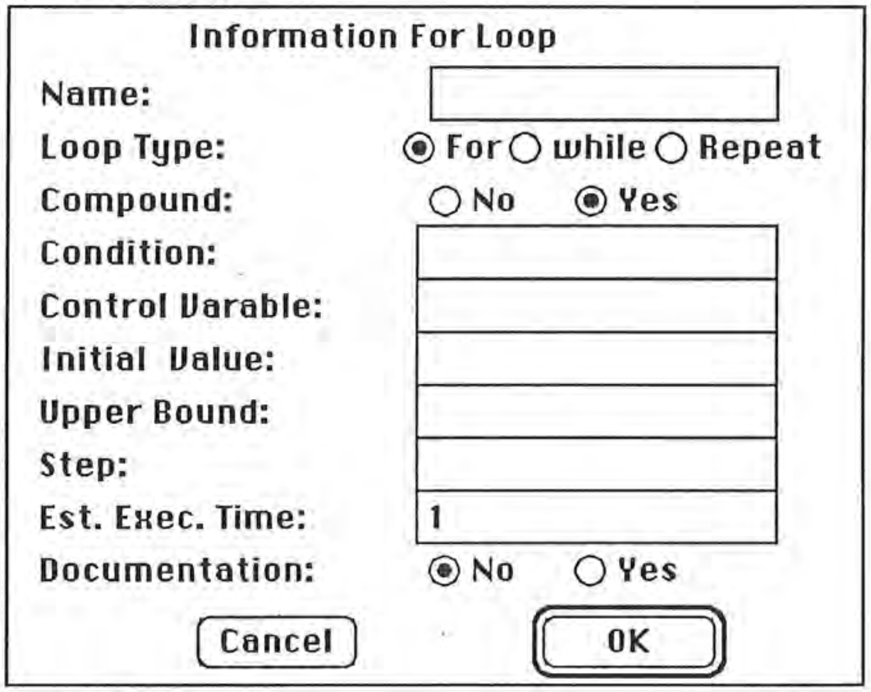


Fig. C.6 Dialog for specifying data type for each variable

6) Loop dialog



Information For Loop

Name:

Loop Type: [✓ For] while Repeat

Compound: No [✓ Yes]

Condition:

Control Varable:

Initial Value:

Upper Bound:

Step:

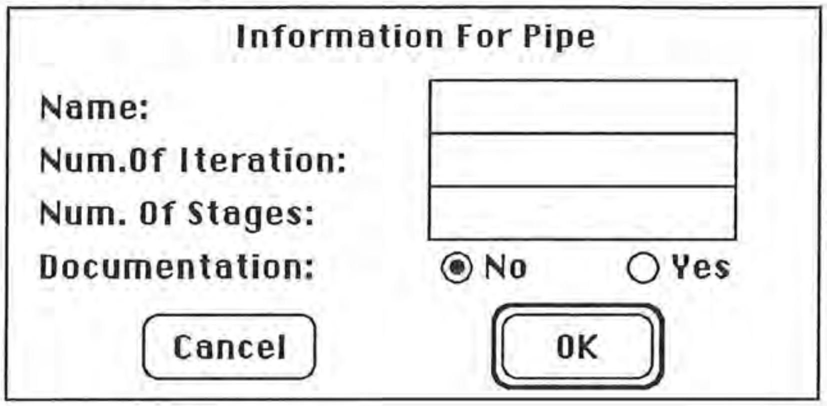
Est. Exec. Time: 1

Documentation: [✓ No] Yes

Fig. C.7 Dialog for Loop

Loop Type: These three radio button lets you specify whether loop is For, while or Repeat.
Condition: This editing session lets you specify the condition of the loop if you select as either while or repeat on loopType session.
Control variable: This session is for entering name of control variable if you select as for loop on loopType session.
Initial Value: This session is for entering initial value of the control variable if you select as for loop on loopType session.
Upper Bound: This session is for entering upper bound value of the control variable if you select as for loop on loopType session.
Step: This session is for entering step value of the control variable if you select as for loop on loopType session.

7) Pipe dialog



Information For Pipe

Name:

Num.Of Iteration:

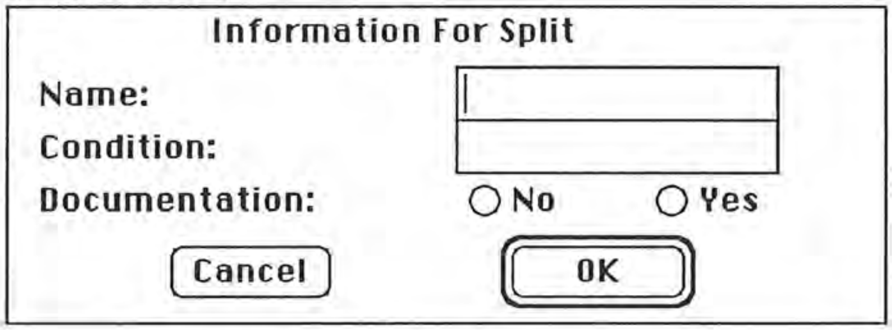
Num. Of Stages:

Documentation: [✓ No] Yes

Fig. C.8 Dialog for Pipe

Number Of Iteration: This editing session lets you specify how many times it should be executed this pipe.
Number Of Stages: This editing session lets you specify how many stages this pipe have.

8) Split dialog



Information For Split

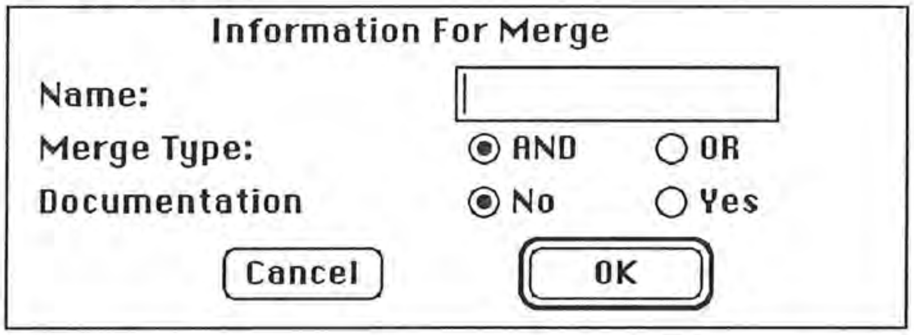
Name:

Condition:

Documentation: No Yes

Fig C.9 Dialog for Split

9) Merge dialog



Information For Merge

Name:

Merge Type: [✓ AND] OR

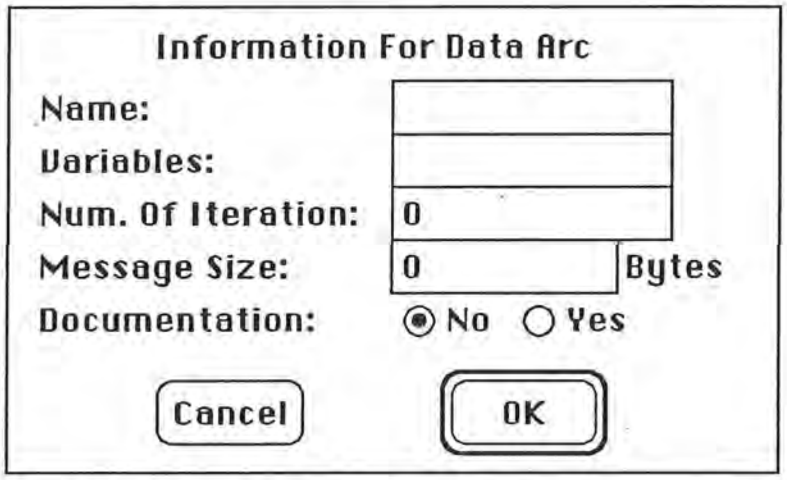
Documentation [✓ No] Yes

Fig. C.10 Dialog for Merge

Merge Type: These two radio button lets you specify whether merge is AND or OR merge.

C.2 Dialogs for entering arc information

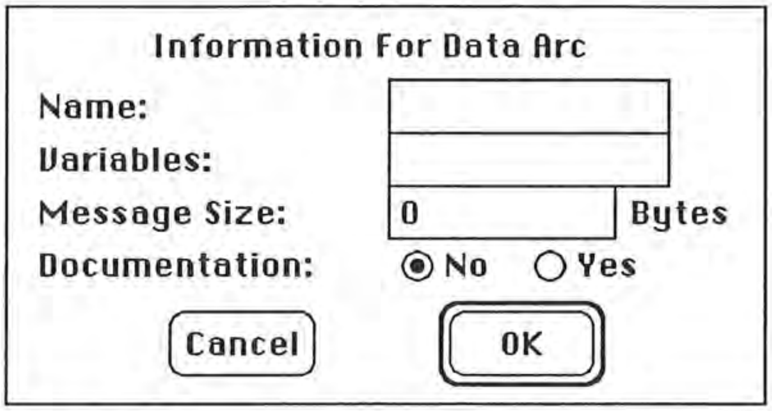
1) Data arc dialogs: Data arc can be either Simple arc or Compound arc. If a data arc connect two simple symbol, it is simple otherwise compound. If it is connected to storage, it should have radio button for expressing mutual exclusion. Therefore, we need four different kind of dialogs and the system automatically show appropriate dialog for each arc. Followings are four different dialogs and brief description is given for each session.


Information For Data Arc

Name:	
Variables:	
Num. Of Iteration:	0
Message Size:	0 Bytes
Documentation:	[✓ No] Yes

Fig C.11 Dialog for Simple Data Arc

Arc Usage: These three radio buttons lets you specify whether arc is to write, read or read/write if it is connected to storage.
Number of Iteration: This editing session lets you specify how many times data will be passed through this arc.
Message Size: This editing session lets you specify how many byte will be passed through this arc.



Information For Data Arc

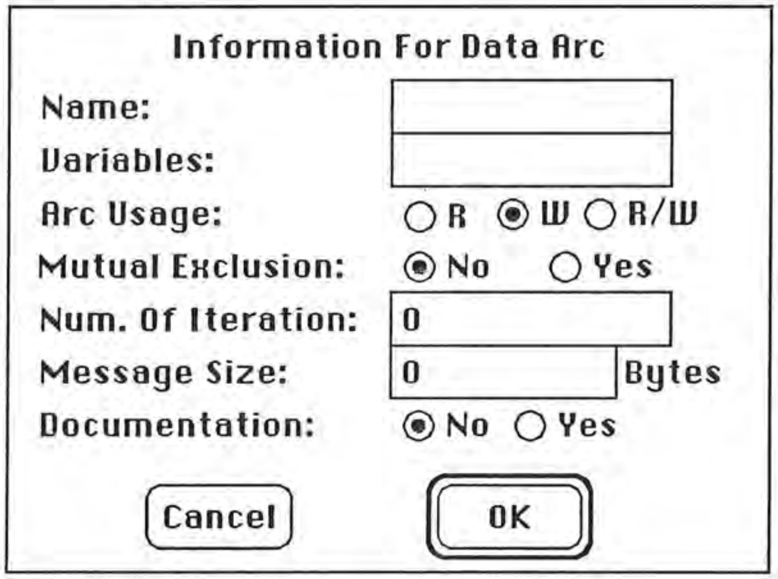
Name:

Variables:

Message Size: 0 Bytes

Documentation: [✓ No] Yes

Fig C.12 Dialog for Compound Data Arc



Information For Data Arc

Name:

Variables:

Arc Usage: R [✓ W] R/W

Mutual Exclusion: [✓ No] Yes

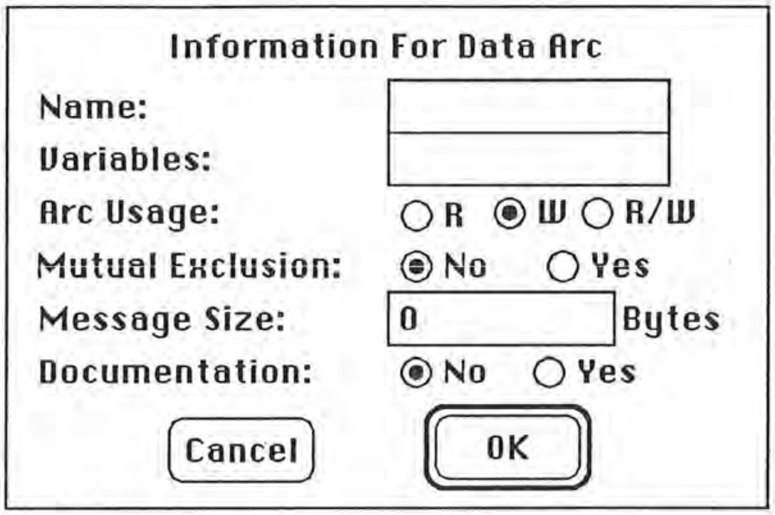
Num. Of Iteration: 0

Message Size: 0 Bytes

Documentation: [✓ No] Yes

Fig C.13 Dialog for Simple Data Arc for Storage

Mutual Exclusion: These two radio buttons lets you specify whether arc is mutually excluded or not.



Information For Data Arc

Name:

Variables:

Arc Usage: R [✓ W] R/W

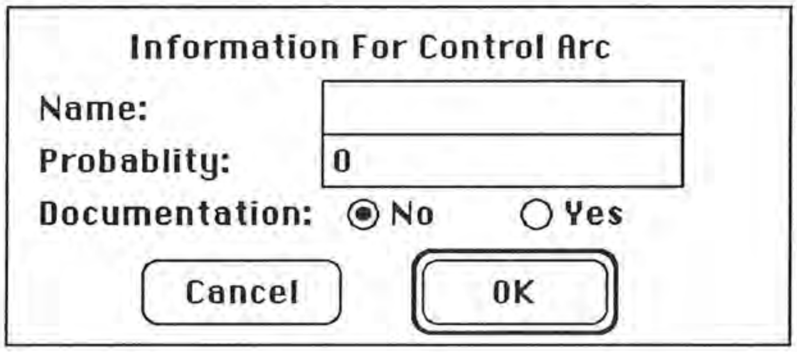
Mutual Exclusion: [✓ No] Yes

Message Size: 0 Bytes

Documentation: [✓ No] Yes

Fig C.14 Dialog for Compound Data Arc for Storage

2) Control Arc dialog:



Information For Control Arc

Name:

Probablity: 0

Documentation: [✓ No] Yes

Fig.C.15 Control Arc

Probability: This editing session lets you specify the probability of this arc will be activated.

References.

1) Apple Computer, Inc.
"Inside Macintosh", May 1986, Volumn I, II.

2) Hesham El-rewini, and Ted Lewis
"Software Development in Parallax:The ELGDF"
Computer Science Department, Oregon State University,
Corvallis Or 97330, 1988.

3) Ted Lewis
"CASE computer Aided Software Engineering"
Computer Science, Oregon State University,
Corvallis, OR 97330

4) Stephen Chernicoff
"Macintosh Revealed", 1987 Volumn I, II.
Hayden Books.

5) OACIS TR-PPSE-89-1
"Parallel Programming Support Environment Research."

6) W.G. Rudd, El-Rewini, Scott Handley, D. V.Judge, Inkyu Kim
Status Report:"Parallel Programming Support Environment
Research at Oregon State University"

7) Daniel D. Gajski, Jih-Kwon Peir
"Essential Issues in Multiprocessor System"
IEEE Software, pp.9-27 June 1985.

8) J.C. Brown, Muhammad Azam, Stephen Sobek
"CODE:A Unified Approach to Parallel Programming"
IEEE Software pp. 10-17 July 1989.

9) Vincent A. Guarna, Jr. Gannon, Jablonowski, Malony, Gaur
"Faust:An Integrated Environment for Parallel Programming"
IEEE Software pp. 20- 26 July 1989.

10) Bill Appelbe, Kevin Smith
"Start/Parallel-Programming Toolkit"
IEEE Software pp. 29- 38 July 1989.

11) Constantine D. Polychronopoulos
Parallel Programming and Compilers.
Kluwer Academic Publisher. 1989

12) Stephen Taylor
"Parallel Logic Programming Techniques"
Prentice Hall. 1989

13) Zhiwei Xu, Kai Hwang
Molecule: A Language Construct for Layered Development of Parallel Programs.
IEEE Transaction on Software Engineering. May 1989.

14) Zary Segall, Larry Rudolph
PIE: A Programming and Instrumentation Environment for Parallel Processing.
IEEE Software November 1985.

15) H. Muhlenbein, O. Kramer, F. Limburger,Streitz.
MUPPET: A programming environment for message-based multiprocessors.
Parallel Computing 8 (1988)P.201-221 North-Holland.
Elsevier Science Publishers B.V.

16) George S. Almasi, Allan Gottlieb
Highly Parallel Computing
The Benjamin/Cummings Publishing Company 1989.

17) Raphael A. Finkel
Large-grain parallelism - Three case studies
The characteristics of Parallel Algorithm.
The MIT Press 1987.

18) James R. McGraw and Timothy S. Axelrod
Exploiting Multiprocessors: Issues and Options
Programming Parallel Processors, Addison-Wesley, Reading,MA

19) Vivek Sarkar
Partitioning and Scheduling Parallel Programs for Multiprocessors
The MIT Press, Cambridge, MA 1989