GENERATION OF MAXIMALLY PARALLEL TASK SYSTEMS

By

KAMALAKAR ANANTHANENI

Bachelor of Science

Kuvempu University

Karnataka, India

1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1997

GENERATION OF MAXIMALLY PARALLEL

TASKS SYSTEMS

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

Blayne E. Mayfield

Jacques C. LaFrance

Wayne B. Powell

Dean of the Graduate College

PREFACE

When tasks are executed in parallel, precedence constraints are placed between mutually interfering tasks in order to ensure the consistency of the results. Sometimes these precedence constraints may be over-specified, in which case the number of tasks executed in parallel may not be maximal. Maximum parallelism can be obtained by removing such unnecessary precedence constraints. The purpose of this thesis was to design and develop a software tool to help study the maximal parallelism extraction algorithm. The tool takes a randomly generated task system as input, checks the task system for determinacy, and generates the corresponding maximally parallel task system for randomly generated domains and ranges. The tool was developed on the Computer Science Department's Sequent Symmetry S/81 computer system running the DYNIX/ptx operating system.

The tool was coded in the C++ programming language using Motif toolkit. The user interface of the tool is based on MotifApp framework. The tool has 37 classes and three major class hierarchies. The results obtained were extensively studied and graphs showing the performance of the maximal parallelism extraction algorithm were drawn for the average, best, and worst cases. It was found that the number of tasks did not play a significant part in the increase in parallelism. The increase in parallelism was found to be generally dependent on the ratio of the number of total memory cells to the number of

domain/range cells. As the ratio increases, the there is more in parallelism in the maximally parallel task system in comparison to the original determinate task system. And, as the ratio decreases, there is a less increase in the degree of parallelism, the worst case being when there is no increase in parallelism. Significant increase in parallelism in the maximally parallel task system was found when the ratio of the number of memory cells to the number of domain/range cells was greater than ten.

ACKNOWLEDGEMENTS

I would like to convey my sincere appreciation to my major advisor Dr. Mansur Samadzadeh for his intelligent supervision, advice, patience, guidance, and constructive criticism. His words of encouragement and moral support are greatly appreciated. My sincere appreciation extends to my other committee members Drs. Blayne E. Mayfield and Jacques LaFrance for serving on my committee. Their time and effort are greatly appreciated.

I would like to give my special appreciation and gratitude to my parents, who always believed in me and my abilities, for their moral and financial support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Due to technological advances and the development of new problem solving methods, significant changes have occurred in the field of computing. There is a need for significant rise in the computing power in almost all fields of science and technology [Fet 95]. Outstanding gains in processing speeds can be achieved through the use of parallel processing [Desrochers 87].

A program in execution is called a process. A process can be represented as a task system. Task systems are generally broken down into subtasks, and these smaller tasks can be further subdivided into threads and executed in parallel on multiple processors or multiplexed on one processor. While executing in parallel, one needs to ensure that the results are consistent, i.e., that each task system is determinate.

In order to make a task system determinate, proper precedence constraints are placed between any two mutually interfering tasks. Sometimes these dependencies may be unnecessary, i.e., precedence constraints may be placed between mutually non-interfering tasks, thus making the task system over-specified. Due to these unnecessary precedence constraints, the maximum possible parallelism may not be utilized.

Maximal parallelism for a task system can be obtained by removing dependencies between mutually non-interfering tasks and adding dependencies between pairs of

mutually interfering tasks. Such a task system, in which the precedence constraints are based only on determinacy requirements, is called a maximally parallel task system and the corresponding precedence graph is the maximally parallel graph.

The main objective of this thesis was to develop a graphical tool which helps in the study of the algorithm for maximal parallelism extraction. Random task systems with random domains and ranges for each task, were used as input to the program. Maximally parallel task systems corresponding to the input task systems were obtained by using Bernstein's conditions [Bernstein 66].

An extensive study of the results was carried out and graphs comparing the degree of parallelism in the original and the maximally parallel task systems were drawn. The tool was implemented using the Motif toolkit and the C++ programming language on the Sequent Symmetry S/81 computer system running the DYNIX/ptx operating system.

Chapter II of the thesis gives a literature review of task systems, determinacy, and maximal parallelism. In Chapter III, the implementation platform and environment are described. In Chapter IV, the design and implementation details of the tool are described. Chapter V gives a summary of the experiments conducted and the results obtained. Finally, Chapter VI summarizes the thesis work and provides suggestions for future work.

CHAPTER II

LITERATURE REVIEW

2.1 Definitions

A task is defined as a unit of computation. When assigned to a processor, a task is executed sequentially [Samadzadeh 92]. A task can be a program, an instruction, or a sub-instruction. A group of tasks can be called a process [Desrochers 87] or sometimes a single task can be a process. A task can be obtained by decomposing or partitioning a given problem into sequential units of computation. The external behavior of a task is defined, but its internal behavior is generally not specified [Coffman and Denning 73].

Depending on the degree of granularity, a task can be a statement (fine-grained partitioning) or it can be a whole program (coarse-grained partitioning) [Samadzadeh 92]. The degree of granularity varies from machine to machine. Generally, a loosely coupled machine is coarsely-grained and a tightly coupled machine is fine grained. Tasks can run in parallel on multiple processors or they can run in a time-shared mode on a uni-processor machine.

A task system can be obtained by partitioning a program. The partitioning can be done either by passing the program through a parallelizing compiler or by using explicit parallel constructs. The code generated by a parallelizing compiler has the advantage of

being portable but it may not be efficient. The programmer typically needs to check such code repeatedly in order for the compiler to recognize the parallelism [Samadzadeh 92]. Using parallel programming constructs requires identification of the independent parts of the program and thus requires a lot of programming experience.

In independent task systems, there are no dependencies between any two tasks, and therefore the tasks can be executed in any order. In a multiprocessor environment the goal is to execute the tasks in such a way as to balance the load on the various processors [Wilson 95]. In a dependent task system the tasks need to be executed in such an order as to ensure the consistency of the results. Any two tasks having dependency between them cannot be executed in parallel.

A task system is a collection of tasks with precedence among them. I/O dependencies associated with the tasks determine the order of execution. A task cannot be executed until all of the tasks on which it depends have been executed. A task system can be represented by the ordered tuple $(\tau, <\cdot)$, where $\tau = \{T_1, T_2, \ldots, T_n\}$ represents the set of tasks and $<\cdot$ represents a partial order among them [Coffman and Denning 73]. The notation $T_1 <\cdot T_2$ indicates that T_1 is a predecessor of T_2 or T_2 is a successor of T_1, and that T_2 cannot start execution until T_1 has finished execution. A terminal task has no successors and an initial task has no predecessors. For a task T_i, $\overline{T}_i$ indicates task initiation and $\underline{T}_i$ indicates task termination.

The memory associated with a task system consists of a set of cells or locations. Associated with each task T is a set of cells called the domain of T, denoted by D_t, and another set of cells called the range of T, denoted by R_t. At initiation a task reads from the

memory cells in its domain and at termination it writes into the memory locations in its range.

The execution of a task system results in the initiation and termination of the various tasks of the task system. For a task system $C = (\tau, <\cdot)$, an execution sequence is defined as $\alpha = e_1, e_2, \ldots, e_{2n}$, where e_i, $1 \leq i \leq 2n$, indicates the initiation or termination of a task in the task system satisfying the following conditions, a. if $e_i = \overline{T}$ and $e_j = \underline{T}$, then $i < j$, and b. if $e_i = \overline{T}_1$ and $e_j = \overline{T}_2$, where $T_1 <\cdot T_2$, then $i < j$. A complete execution sequence starts with the initiation of the first task and ends with the termination of the final task. If α is an execution sequence, then $V(M_i, \alpha)$ indicates the sequence of values written into the memory location M_i, $1 \leq i \leq m$, for a memory of size m, during the execution sequence.

2.2 Modeling of Task Systems

Graph models can be used to represent task systems. They represent the data dependency and structure of the computation for a given problem. Directed acyclic graphs (DAG), data flow graphs, and Petri nets are examples of some of the widely used graph models.

When studying and detecting parallelism in a program, a data flow graph of the program is first drawn, and converted into a DAG using loop removal algorithms. These DAGs are then converted to precedence graphs by removing the redundant paths [Samadzadeh 92]. Precedence graphs, as the name suggests, depict precedence constraints and help determine the order of execution of the tasks. They represent the cooperation

and communication among the tasks in a task system, and hence aid in representing and detecting the parallelism inherent in a task system.

A precedence graph consists of a set of nodes and a set of edges connecting the nodes. The nodes represent the tasks and the edges represent precedence among the tasks. An edge from task T_1 to task T_2 indicates that T_1 is an immediate predecessor of T_2, and that T_1 should terminate before T_2 is initiated. Tasks T_1 and T_2 can be executed in parallel if T_1 is not a successor nor a predecessor of T_2. An independent task system does not have any dependencies among tasks, so there are no edges in the graph representation of the task system.

An important use of precedence graphs is to extract the parallelism inherent in a task system and to ensure the consistency of the results by selectively imposing appropriate precedence constraints between pairs of tasks in under-specified task systems. The width of a precedence graph is an indication of the maximum amount of parallelism possible in that task system. Scheduling algorithms take into account the precedence imposed by precedence graphs, and schedule the tasks in such a way as to obtain the shortest schedule length and to ensure the determinacy of the system.

An adjacency matrix can be used to represent the connectivity of the nodes in a task system. The row and column headers of an adjacency matrix represent the nodes, and the entries of the matrix represent the connectivity among the nodes.

2.3 Determinacy of Task Systems

When a program is broken down into tasks that can be executed in parallel, an important issue of concern is the consistency of the results obtained. In a multiprocessor

environment the speeds of different processors may vary. A task system is said to be *determinate* if the results produced by it for a specific input are unique regardless of the execution sequence or the speed of execution [Coffman and Denning 73]. A *non-determinate* task system may execute in different ways, for various given inputs and give different results [Wilson 95]. Most serial computations are necessarily determinate whereas parallel computations may fail to be determinate due to the timing effects and the allocation of data to various processors.

Formally, a task system is said to be determinate if, for a given initial state S0, and any pair of execution sequences α and α' of the task system, $V(M_i, \alpha) = V(M_i, \alpha')$, $1 \leq i \leq n$, where n is the number of memory cells [Nutt 92] [Coffman and Denning 73]. $V(M_i, \alpha)$ and $V(M_i, \alpha')$ represent the value sequences stored in memory cell M_i for execution sequences α and α', respectively. Bernstein [Bernstein 66] derived a set of conditions that determine whether two parts of a program can be executed in parallel. He basically stated that tasks with overlapping domains and ranges cannot be executed in parallel.

A task system consisting only of mutually non-interfering tasks is determinate. Two tasks T and T' are said to be *mutually non-interfering* if a. T is a successor or predecessor of T', or b. if T and T' satisfy Bernstein's conditions, i.e., i. $D_T \cap R_{T'} = \Phi$, ii. $R_T \cap D_{T'} = \Phi$, and iii. $R_T \cap R_{T'} = \Phi$. Independent task systems are determinate since they do not exhibit data dependencies. Dependent task systems share memory cells and hence are not generally determinate. In order to ensure determinacy, proper precedence constraints should be introduced between pairs of dependent tasks where necessary.

2.4 Maximal Parallelism

Precedence plays an important role in task systems since, among other things, it helps solve the Readers-Writers problem [Nutt 92]. A task reading from a memory location can be run in parallel with other tasks reading from the same location but not with a task writing into the same memory location. A task writing into a memory location cannot be run in parallel with any other task accessing the same location.

A task system may be over-specified in which case there may be too many precedence constraints, or it may be under-specified in which case there may not be enough precedence constraints. An under-specified task system may not be determinate; and, in the case of over-specified task systems, the number of tasks that can be executed in parallel may not be maximal. A maximally parallel task system is obtained from a determinate over-specified task system by removing unnecessary precedence constraints while preserving determinacy.

In a task system, precedences may be introduced due to some external reasons thus making it over-specified [Nutt 92]. This reduces the amount of potential parallelism in the task system. Maximal parallelism can be obtained by removing such unnecessary precedences. A task system is said to be maximally parallel if the removal of precedence between any two tasks makes them interfering, i.e., if G is the graph representing the maximally parallel task system, then the removal of an arc between any two nodes will make the corresponding tasks interfering [Coffman and Denning 73].

In a maximally parallel task system, precedence constraints are based only on shared memory references, i.e., they are based only on determinacy constraints. When obtaining the maximum possible parallelism, first all tasks are considered to execute in

parallel and then precedence constraints are imposed on any pair of tasks violating Bernstein's conditions (see Section 2.3) [Bernstein 66].

CHAPTER III

IMPLEMENTATION PLATFORM AND ENVIRONMENT

3.1 Sequent Symmetry S/81

The Sequent Symmetry S/81 machine, developed by Sequent Computer Systems, Inc., is a tightly coupled multiprocessor that uses shared memory. It has a parallel architecture with multiple processors, with DYNIX/ptx or DYNIX V3.0 operating system and standard interfaces such as Ethernet, MULTIBUS, SCSI, and VMEbus [Sequent 90]. UNIX compatible software can be run on Symmetry S/81 with little or no modifications.

DYNIX/ptx is compatible with AT&T SystemV3.2 only, where as DYNIX V3.0 supports both Berkeley UNIX and UNIX System V command sets [Sequent 90]. DYNIX/ptx is a flavor of UNIX that dynamically distributes the various responsibilities such as handling interrupts to all processors in the system.

3.2 X Window System

The X Window system and its library provide the basic functions needed to implement graphical user interfaces. It provides a wide range of capabilities such as window creation and drawing, event handling, inter-client communication, and input/output buffering.

The architecture of the X Window System is based on the *client-server* model. The client is the application program and the server is the X terminal. The server packages the user's action into an event and sends it to the client which in turn sends commands to the server [Brain 92]. The terminal then interprets these commands and updates its display. The keyboard, screen, and the mouse together form the X terminal's display. The server hides the differences in the underlying hardware from the client application [Nye 90]. The X protocol (see Section 3.3) helps in the communication between the client and the server.

The server's primary responsibility is to manipulate and display windows as requested by the client. Windows in X form a hierarchical structure. Initially, when the X server starts, it creates the root window. All other windows are descendants of the root window [Young 92]. When a window is created, it is not visible until it is mapped. Windows can be displayed or removed from the screen by mapping or unmapping them, respectively. Windows are created at the request of the client and are no longer visible after the client disconnects.

Figure 1 shows the Motif/Xt/X/UNIX library hierarchy [Brain 92]. At the bottom there is UNIX, and on top of it is X and its libraries. The X toolkit sits on top of X, and above it is Motif and its library.

The X toolkit consists of two parts: the Xt Intrinsics and a set of user interface components called widgets [Johnson 90]. Xt Intrinsics supports many widget sets such as the Athena widget set, the HP widget set, the Open Look widget set, and the Motif widget set. These widget sets can be used interchangeably. The Motif widget set was used in the development of the tool for the thesis work.

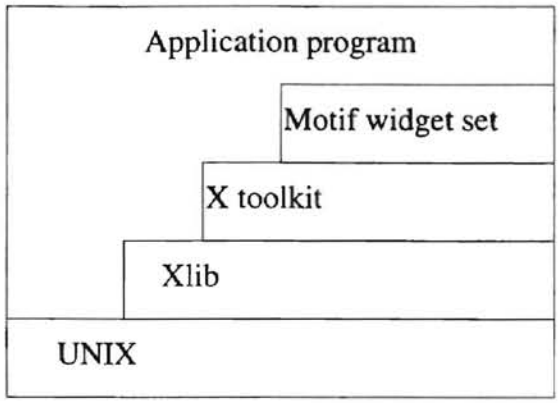


Figure 1. Motif/Xt/X/UNIX Hierarchy (Source: [Brain 92])

3.3 X Protocol

The X protocol is a true definition of the X window system and is responsible for communication between the client and the server [Nye 90]. It operates asynchronously by using a single bi-directional network connection. It multiplexes all the windows on the same screen thus preserving the order of the events.

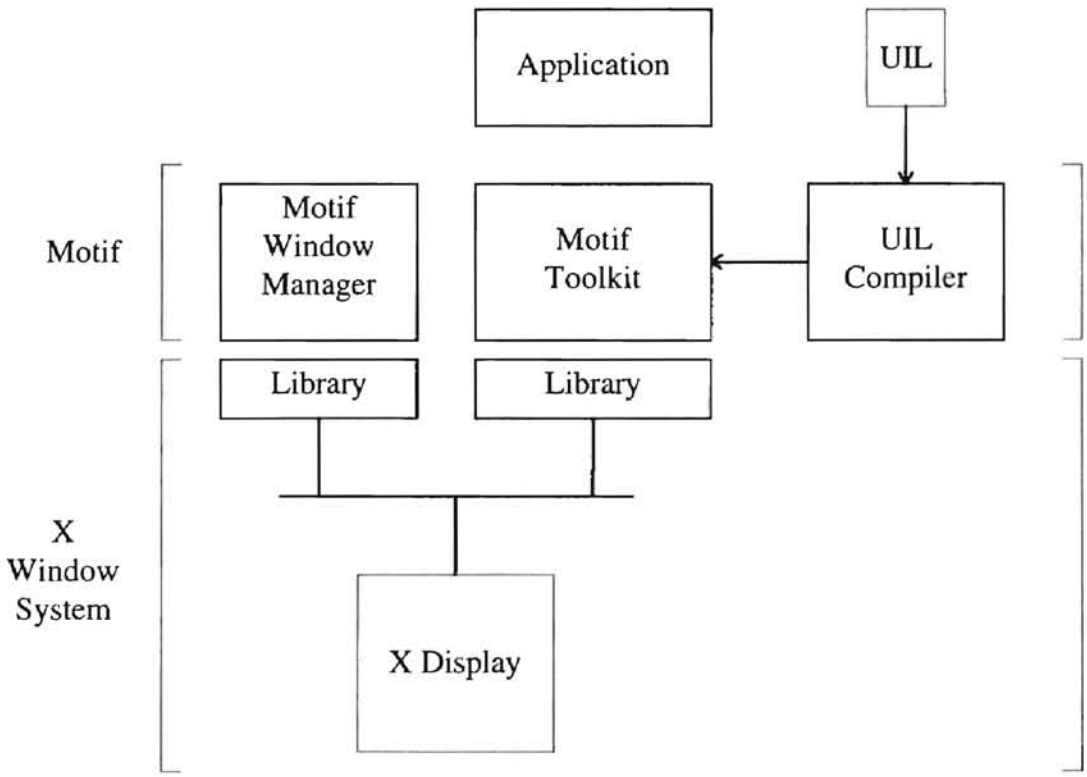
Below the X protocol any lower network layer can be used. The clients implement the X protocol that interfaces with the underlying network protocol [Nye 90]. Servers can understand more than one network protocol, thus allowing them to communicate with clients in more than one network at once. The connection between a client and a server on the same machine is based on local interprocess communication (IPC) channels, UNIX domain sockets, or shared memory.

X can talk to any language provided that a programming interface exists between the language and the X protocol [Brain 92]. Xlib is the library that implements the X protocol for the C programming language.

3.4 OSF/Motif Toolkit

The OSF/MOTIF is a standard user interface toolkit that was developed by the Open Software Foundation and is based on Xt Intrinsics [Young 92]. Motif was developed to provide the functionality needed to implement graphical user interfaces that work alike on a wide variety of platforms [Berlage 91].

Motif provides a set of user interface components called widgets, which enable programmers to include the most frequently used graphical interface elements in their programs. Motif consists of the Motif Widget Set, the Motif Style Guide, the User Interface Language and the Motif Window Manager [Heller 94]. Figure 2 illustrates the architecture of Motif.



Legend:
UIL: User Interface Language

Figure 2. The architecture of OSF/Motif (Source: [Berlage91])

In the layered architecture of OSF/MOTIF, the X Window System acts as the lower layer, making the system machine independent . X provides the basic event driven programming environment. The X library (Xlib) provides the programming interface to the X Window System. X imposes no restrictions on the appearance of the user interface components, but the user needs to take care of all the low level event handling, thus making the programs long and complex.

The Motif window manager, mwm, manages the basic functions of a window such as resizing, moving, and iconizing, thus reducing the amount of code to be written by the programmer. The mwm follows a specific protocol specified in the Inter-Client Communications Conventions Manual (ICCCM). Any ICCCM complaint window manager can be used with X and Motif applications [Young 92].

The Motif toolkit is based on Xt Intrinsics and provides a set of widgets, which take care of all the low level event handling for the programmer. Motif provides basic widgets such as text widgets, scrollbar widgets, and push button widgets. Motif also provides manager widgets such as form widgets, which control the layout of other widgets. Unlike X, Motif enforces a particular look and feel for the user interface components, thus maintaining consistency of the user interface and at the same time restricting the freedom available in X.

In Motif programming, each widget has a set of resources that control its appearance and behavior. The resources of a widget can be read or set to new values. For example, to change the height and width of a push button widget, its height and width resources need to be set. The widget can also send out messages called *callbacks* to communicate with your code.

The user interface language (UIL) is a text based language used to describe a user interface consisting of Motif or other widget sets. UIL provides a simple syntax for specifying the user interface in terms of a widget hierarchy [Heller 94]. UIL can be used as an alternative to the C language interface for applications based on Motif. Using UIL, an application's window layout can be described in a file separate from the rest of the program [Young 92].

The Motif style guide describes the preferred ways to design a Motif application interface, to establish consistency among Motif applications. Even though a certain amount of consistency is ensured by the use of the Motif toolkit, there are some guidelines which cannot be enforced and many which can be overridden [Berlage 91]. Examples of such guidelines include changing the cursor shapes to inform the user about what is happening and providing some standard menu entries which should be used in the same place in all applications that need them [Brain 92].

3.5 Using Motif with C++

Motif, like C++, supports an object-oriented architecture, but the object-oriented architecture of Motif is incompatible with the class hierarchy of C++ [Young 92] [Bowman 95]. One cannot create a C++ class as a subclass of a motif widget. One approach is to wrap C++ classes around widgets [Bowman 95]. Another approach, as suggested by Young [Young 92], is to put together one or more widgets into a logical grouping. The widget creation, resource specification, and assigning callbacks can all be put in a C++ class.

Callbacks provide a means to notify the program of a change or a user action. Callbacks present a problem for C++ classes. C++ member functions require a hidden first argument, which is the *this* pointer. If a C++ member function is called from a C function as a callback, the *this* argument will not be supplied, resulting in an incorrect argument sequence [Young 92] [Bowman 95]. This problem can be solved by having an external function as the callback routine and having it call the appropriate member function. A more simple and direct approach is to use static member functions. A static member function does not accept a *this* argument and C-based Motif functions can call these functions directly [Young 92] [Bowman 95].

CHAPTER IV

DESIGN AND IMPLEMENTATION

This chapter describes the design and implementation issues of the program (i.e., the software tool created as part of this thesis). The class hierarchies, application framework used, and other related issues are described in this chapter. The program has been implemented using C++ and OSF/Motif. There are 39 classes, the class headers are declared in the file class.h, and the implementations are declared in the file class.C, where *class* stands for each class name.

4.1 Overview of the Tool

The program was developed as a tool that aids in the study of the maximum parallelism extraction process. The program takes random task systems together with random domains and ranges for the tasks using the random number generator provided by Park and Miller [Park and Miller 88]. The task systems are made determinate by placing dependencies between mutually interfering tasks and removing redundant dependencies. The maximally parallel task system is obtained from the determinate task system by placing dependencies between mutually interfering tasks and removing all other dependencies.

The inputs to the program are random task systems obtained by using the random task system generator developed by Samadzadeh [Samadzadeh 92]. There are two formats for input: one is the fixed format in which a task system and its domains and ranges are provided, and the other is the random format in which only the task system is provided. For fixed format input files, the results of the program can be predicted since ranges and domains are provided by the user. In the case of random format data files, the results cannot be predicted since the program generates random domains and ranges, and obtains the maximally parallel task system based on them.

Figure 3 illustrates a random format input file, where N indicates the number of tasks. Task systems are represented by means of upper triangular adjacency matrices. In a task system generated by the random task generator [Samadzadeh 92], the higher numbered tasks can depend only on the lower numbered tasks, hence an upper triangular matrix is sufficient to represent the task system. For example, task 1 can depend on task 0 and not vice versa. No domains or ranges are provided in the random format input file.

Figure 4 illustrates a fixed format input file. In the fixed format input file, the domains and ranges are specified after the task system, as shown in the figure. In the figure, 25 indicates the upper limit on the number of memory cells. So memory has cells ranging from 1 to 25. The number of lines of domains and ranges must be equal to the number of tasks. Since there are ten tasks in the task system in Figure 4, ten domains and ranges are provided. Each line contains the number of domain cells followed by the domains cells and the number of range cells followed by the range cells, exactly in that order. The domains and ranges of the lower numbered tasks precede the domains and ranges of the higher numbered tasks. For example, line 4 (0,1,2,3) 2 (0,1) represents the

domains and ranges of task 0 (tasks are numbered from 0). This line specifies that task 0 has four domain cells (0, 1, 2 and 3) and two range cells (0 and 1).

```
N = 10
0110000000
 001100010
  01101101
   0001100
    011010
     01100
      0111
       001
        01
         0
```

Figure 3. Example of a random format input file

```
N = 10
0110000000
 001111010
  01001011
   0100001
    010010
     01101
      0101
       001
        01
         0

25
4 (0,1,2,3) 2 (0,1)
3 (0,4,5) 3 (7,8,9)
3 (8,9,0) 3 (12,13,14)
3 (6,13,20) 4 (0,7,10,19)
3 (8,9,11) 2 (6,12)
4 (13,14,15,16) 3 (9,11,20)
2 (18,20) 1 (18)
0 () 1 (18)
4 (0,1,2,3) 2 (3,18)
2 (3,5) 1 (0)
```

Figure 4. Example of a fixed format input file

A graphical user interface is provided that enables users to view the precedence graphs, adjacency matrices, and comparisons between the determinate and maximally parallel task systems. A help menu containing explanation about running the program and the input to the program is also provided.

4.2 Class Structure

The algorithm part of the program consists of six main classes: TaskSystem, Task, Memory, RandGen, Dependency, and IntArray. The TaskSystem class is the backbone of the program. It has pointers to Task and Dependency classes as data members and they represent the tasks and the dependencies between them. The TaskSystem class has member functions checkRedundancy() and rmRedundancy() for removing redundant dependencies, makeDeterminate() to make the task system determinate, degOfParallelism() to find the degree of parallelism of the task system, and getMaxParallelism() to make the task system maximally parallel.

The Task class contains the task number and an instance of Memory class as its data members. The Memory class represents the memory associated with a task and has data members _domain and _range, representing the domain and range of the task, respectively. The _domain and _range are IntArray instances. The Dependency class contains private data members _taskId, _dependsOn, and _dependents specifying the tasks on which the task depends and the tasks which depend on the task. Both _dependsOn and _dependents are IntArray instances.

The RandGen class is an implementation of the random number generator provided by Park and Miller [Park and Miller 88] and is used to generate the random

domains and ranges associated with a task. The RandGen class is used for generating domains and ranges when the input is random format file. In case of the fixed format files, since the user provides domains and ranges, the random number generator is not used. The IntArray class represents an integer array and contains the needed member functions to manipulate arrays. The IntArray class has overloaded addition, subtraction, equality, and subscription operators defined for integer arrays.

4.3 User Interface

4.3.1 Application Framework

An application framework is a collection of pre-existing, and tested reusable classes that captures features common to many applications [Young 92]. It provides the base design on which new applications can be built. This frees the programmer of the burden of connecting the various components of the program, since the application framework does this automatically. Application specific details can be added by adding new methods, or by providing new classes that derive from classes provided by the framework.

Application frameworks are different from toolkits [Young 92]. Toolkits (like Motif) provide functions that a programmer can use in a program. The programmer has the freedom in deciding how to use them. An application framework, on the other hand, is more restrictive, provides more structure to the program and gives less freedom to the programmer. One such application framework is the MotifApp framework described in the next subsection.

4.3.2 MotifApp Framework

The user interface for the tool was developed based on the MotifApp framework provided by Young [Young 92]. This framework captures features common to most Motif applications, such as:

- Initializing the Xt Intrinsics
- Opening a connection to the X server
- Creating an application context (XtAppContext)
- Creating a shell widget to serve as the parent for other widgets
- Creating one or more widgets that define the user interface
- Handling events by entering an event loop

The framework uses C++ classes to combine groups of widgets into more complex user interface components and to define the overall behavior of the user interface components. The user interface components of MotifApp framework have the following features [Young 92]:

- Components create one or more widgets in the class constructor. Normally, callbacks and other setup are handled here as well. Each component creates a single widget that forms the root of the widget tree represented by the class. All other widgets are children or descendants of this base widget.
- Components take a widget and a string as arguments in the constructor. The widget serves as the root of the component's widget tree and the string is used as the name of the root widget.
- Each component class provides an access method that can be used to retrieve the root widget of the component's subtree.

- Components allow the widget subtree encapsulated by the class to be managed and unmanaged. Components are treated as a logical group. Other widgets are managed in their constructors and only the root widget needs to be managed or unmanaged.
- Components handle the destruction of widgets within the components widget tree. The widgets encapsulated by an object should be destroyed when the object is destroyed.

4.3.3 High Level User Interface Components

Figure 5 illustrates the high level User Interface hierarchy. The BaseWidget class serves as the base class for all user interface components. It consists of a data member _baseWidget that acts as the root of all widget trees. The derived classes create the base widget by calling BaseWidget class's constructor. It has two member functions manage() and unmanage() to manage and unmanage the component's base widget. The BaseWidget is intended specifically to serve as the base class for other classes and cannot be instantiated. Sections 4.4 to 4.8 contain description of the various high level user interface components.

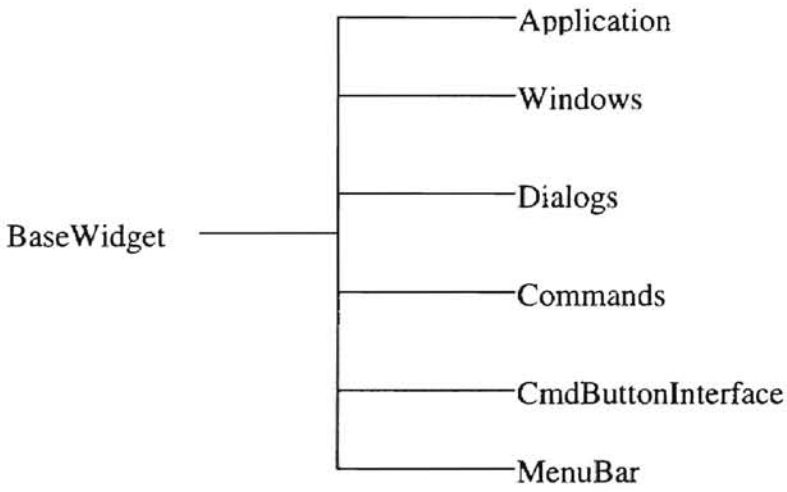


Figure 5. Higher level User Interface hierarchy

The Application class in Figure 5 implements the features common to most motif applications and provides a place to store data that may be needed throughout an application. The Application class handles X/Xt initialization, handles events by entering the event loop, creates a main shell that serves as the parent for all other top-level windows, maintains global data structures such as the X display and application context, and provides means to manage, unmanage, and iconify all the windows registered with it.

The Application class has the member function initialize() that initializes the XtIntrinsics, creates the Application object's base widget, creates an application context, opens a connection to the X server, and initializes all the main windows in the application. The Application class provides a global pointer to an instance of the Application class, theApplication, which must be instantiated once for each application and which serves as the parent widget for all top-level windows.

4.3.4 Windows

The tool has options to display the graphs and the adjacency matrices of the determinate and the maximally parallel task system, as well as the results of the execution in terms of the possible improvement in the degree of parallelism. Figure 6 shows the class inheritance hierarchies of the windows in the program.

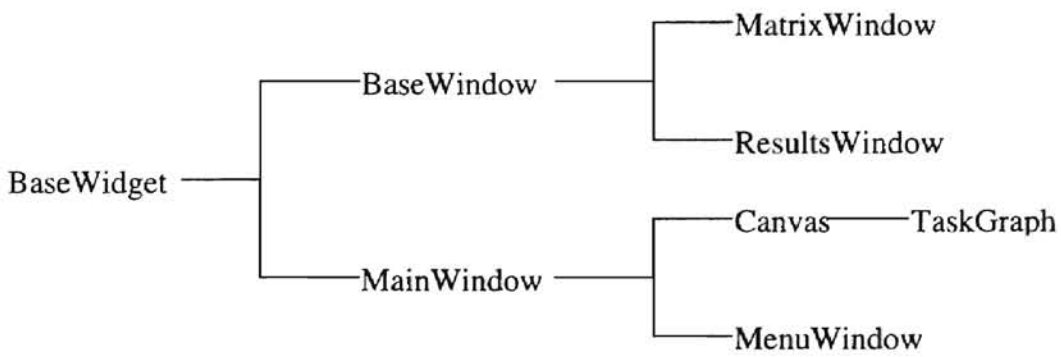


Figure 6. Class Hierarchy of the windows

The following paragraphs contain descriptions of the classes representing the various windows used in the program:

The BaseWindow class is an abstract class derived from the BaseWidget class and provides all the derived classes with a base window along with the base widget provided by the BaseWidget class. It has pure virtual functions createShell() and createBaseWidget() which the derived classes need to override in order to form the base widget and the base windows, respectively. It has a virtual function initialize() that is called from the initialize() function in the derived classes. The initialize() function calls createShell() and createBaseWidget() functions in the derived classes to create the top level shell and to create the base widget.

The MatrixWindow class is derived from the BaseWindow class and displays the adjacency matrices of the determinate and the maximally parallel task systems. It uses a label widget to display the matrices and also provides an OK button, which when pressed unmanages the window. Figures 7 and 8 show snapshots of the MatrixWindow for a determinate task system and the corresponding maximally parallel task system.

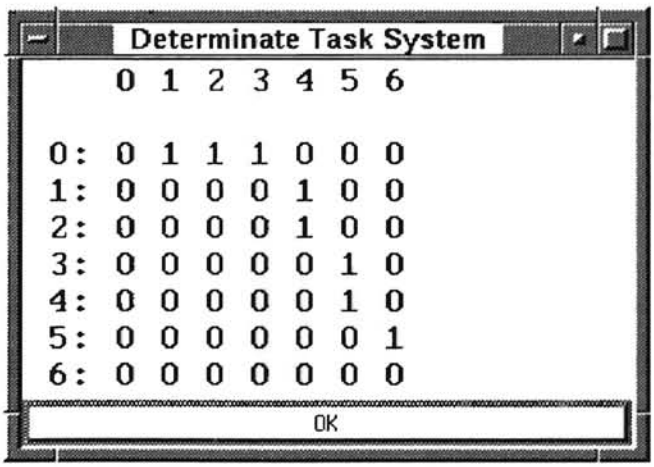


Figure 7. Adjacency Matrix of a determinate task system

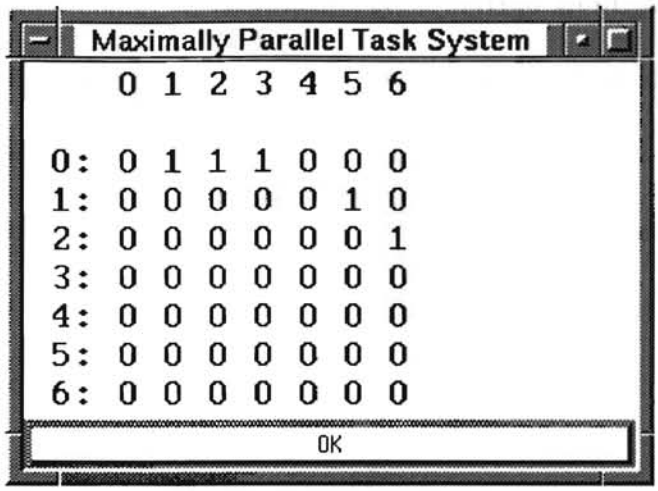


Figure 8. Adjacency Matrix of a maximally parallel task system

The ResultsWindow class displays the comparison of the determinate task system and the maximally parallel task system. Its class structure and operation are similar to the MatrixWindow. Figure 9 gives a snapshot of the Results window.

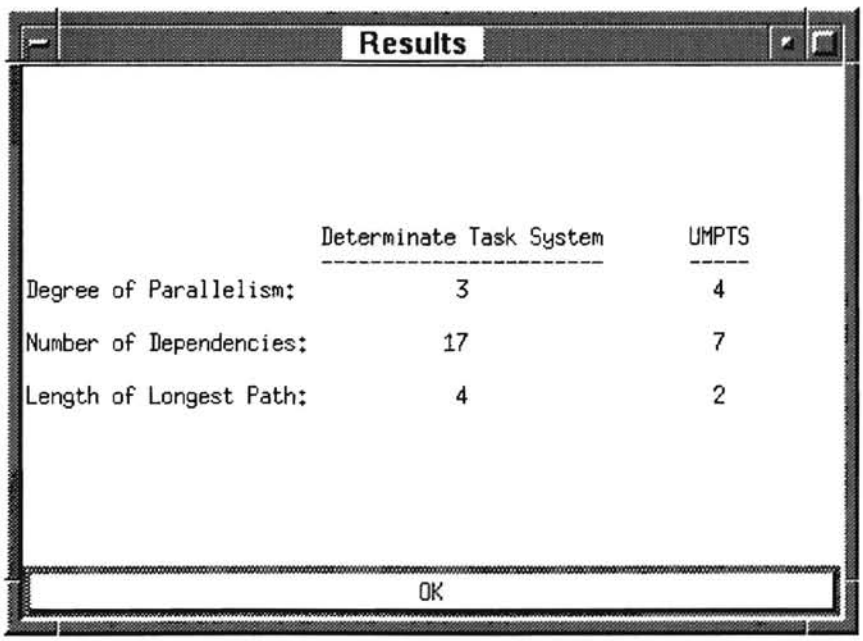


Figure 9. A snapshot of the Results window

The MainWindow class is an abstract class derived from the BaseWidget class and provides a main window widget. The MainWindow class provides the layout of the application's top-level window and registers with the application object by calling the registerWindow() function. The initialize() member function creates the top-level shell and the main window. The manage() and unmanage() member functions are responsible for mapping and unmapping the window.

The Canvas class is derived from the MainWindow class and is the base class of the TaskGraph class. It has a drawing area widget, a graphics context, and pointers to line and node classes as protected members so that derived classes can use them. It has pure virtual functions setPoints() and draw() which are implemented by the TaskGraph class. It has a member function initialize() called from TaskGraph() which in turn calls the initialize() member function in the MainWindow class. The initialize() member function creates a drawing area widget and sets up the graphics context.

The TaskGraph class is used to draw the precedence graphs of both the determinate and the maximally parallel task systems. It is derived from the Canvas class and draws the graphs on the drawing area widget available in the protected area of Canvas. It has function setPoints(), to set the points of the nodes and lines of the precedence graph, and function draw(), to initialize the drawing area and to register an expose callback for the drawing area; so that whenever the widget is exposed, the precedence graph is redrawn. Figures 11 and 12 show precedence graphs of a determinate task system and its corresponding maximally parallel task system.

The MenuWindow class is derived from the MainWindow class and provides a menubar. All of the commands used in the program are present as the protected data members of the MenuWindow class and are created in its constructor. The MenuWindow class has a member function initialize() that initializes the main window and then creates a menubar. The commands are then added to the menubar in the function createMenuPanes() called from the initialize() member function. Figure 13 shows the main window and the menu bar created when the program is run.

4.3.5 Dialogs

Dialogs can be used to provide information to the user or to request information form the user. The dialogs in the program are derived from the BaseWidget, as shown in Figure 10. There are two types of dialog classes, one is the DialogManager class that provides error information and Question dialogs for presenting information and getting feedback from the user, and the other is the FileSelectionDialog class that provides a file selection box that prompts the user to enter an input filename or select a file from the list of filenames displayed .

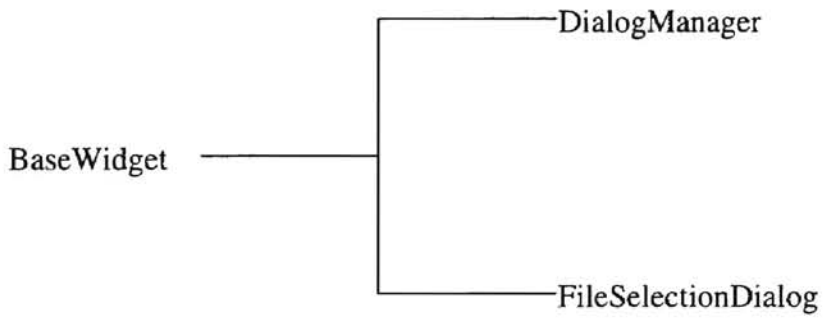


Figure 10. Class hierarchy for the dialog

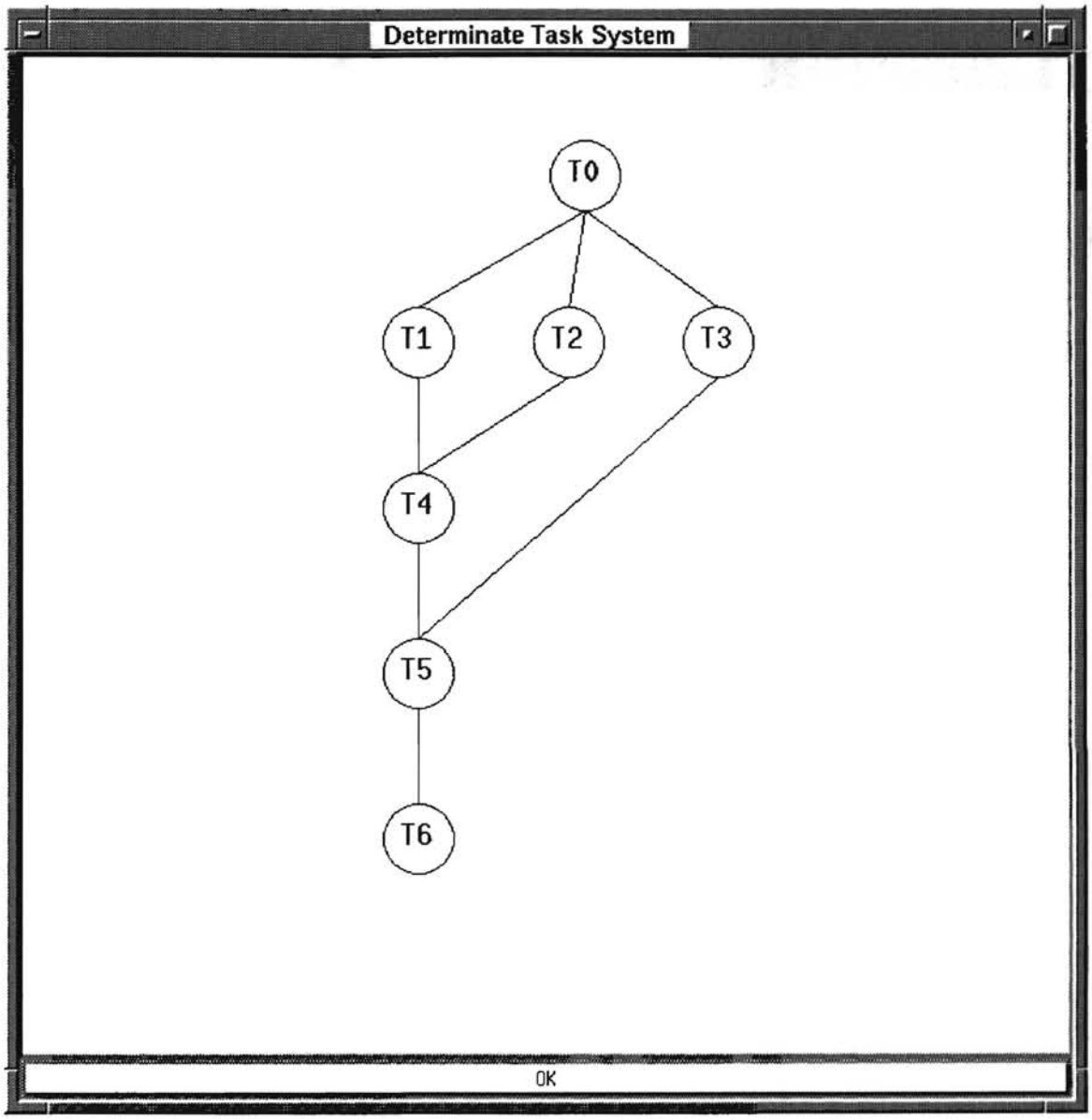


Figure 11. Precedence graph of a determinate task system

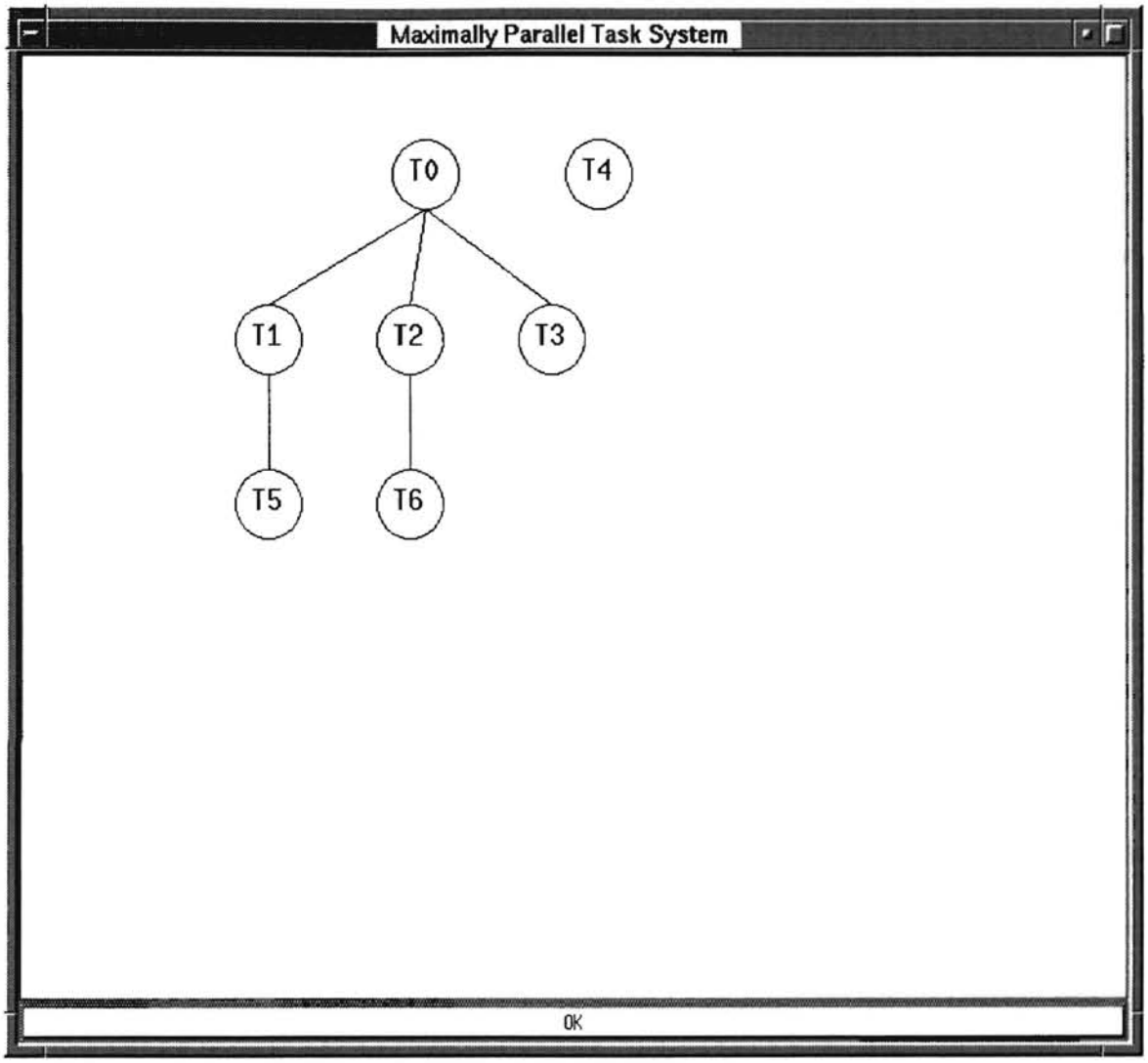


Figure 12. The precedence graph of the maximally parallel task system corresponding to the determinate task system in Figure 8

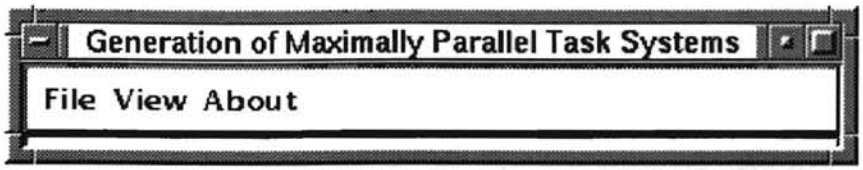


Figure 13. Main Window and the Menu bar

4.3.5.1 DialogManager

The DialogManager class is derived from the BaseWidget class and supports popup dialogs. It supports information, question, and error dialogs. The constructor of the DialogManager class takes two arguments, the first one indicates the name of the dialog and the second one indicates the type of the dialog. Depending on the type of dialog specified, an error, a question, or an information dialog is created. The error dialogs inform the user of an erroneous situation. Figure 14 illustrates an error dialog box. The information dialog can be used to supply information to the user. Figure 15 gives a snapshot of a sample information dialog box. The question dialog box is used to ask a question from the user. For example, the question dialog in Figure 16 is used to confirm the exit operation. The DialogManager class supports global variables theInfoDialogManger, theErrorDialogManager, and theQuestionDialogManager that are instantiated once and can be used throughout the program.

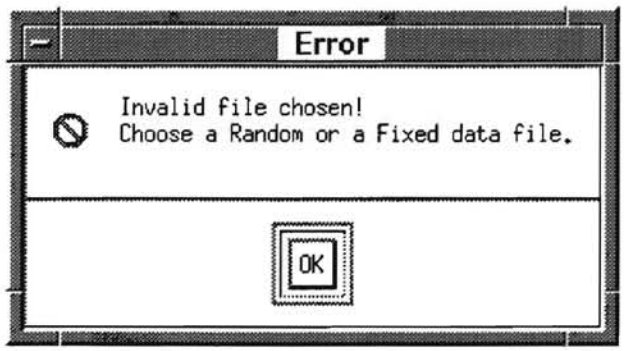


Figure 14. Error Dialog Box

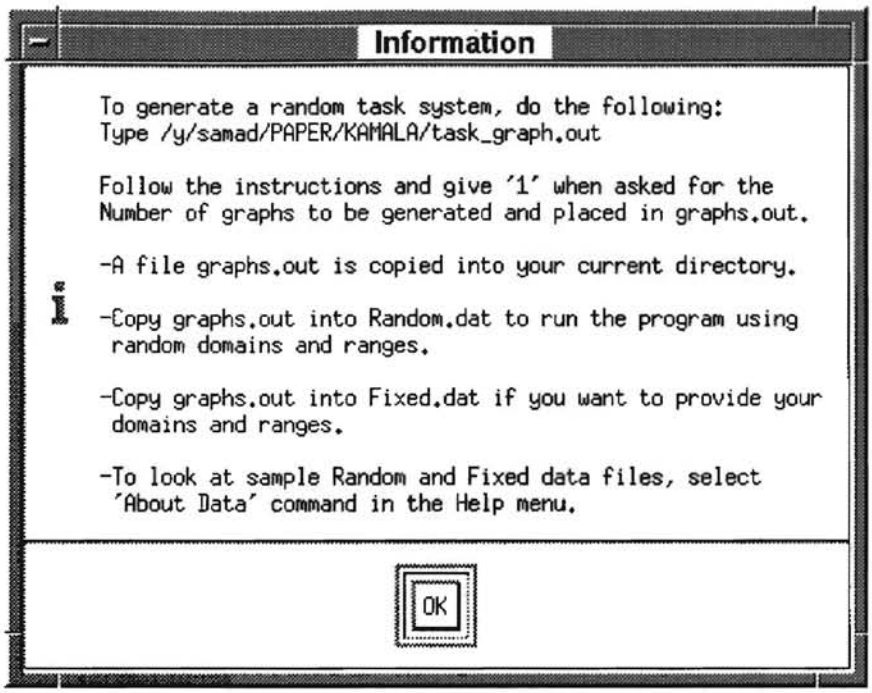


Figure 15. Information Dialog Box

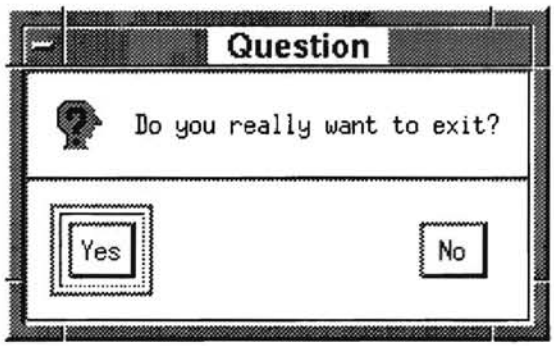


Figure 16. Question Dialog Box

4.3.5.2 FileSelectionDialog

The FileSelectionDialog class provides a file selection dialog box. A file selection dialog box is created and displayed when the user selects the Open command in the File menu. The user is prompted to enter or select a file name. Then the file selected by the user is validated. If it is invalid, an error is displayed using the theErrorDialogManager. If the file selected is valid, the file name is passed to the readFileCallback() function in the OpenCmd class. Figure 17 illustrates a Motif file selection dialog box.

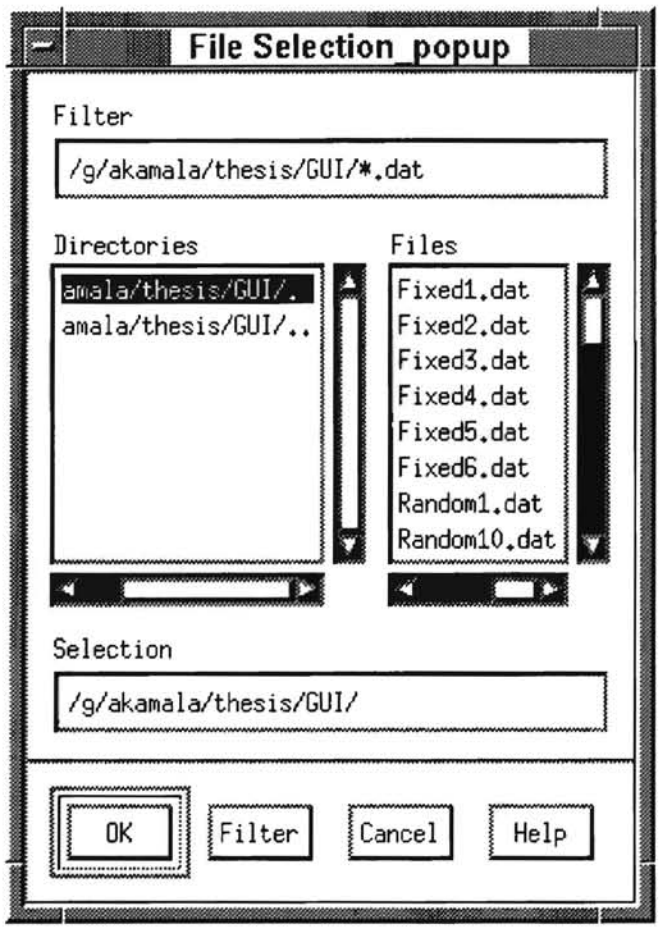


Figure 17. File Selection Dialog Box

4.3.6 Commands

Commands represent user actions and each command is represented as a class. The Cmd class is an abstract class that acts as the base class of all the commands used in the program, and contains information common to all of the commands. These commands are added to the menubar so that they can be selected by the user. The Cmd class has

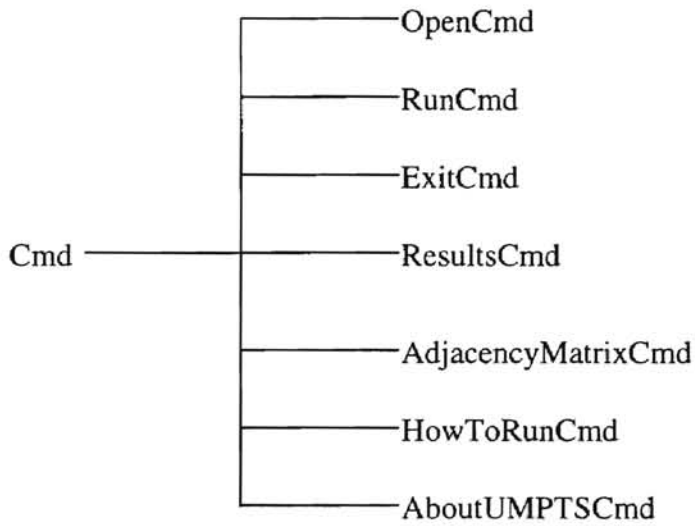


Figure 18. Cmd class hierarchy

member functions to activate and deactivate a command. The Cmd class has a flag _active indicating if the command is currently active. It also contains private members _activationlist and _deactivationList specifying the list of commands to be activated and deactivated, respectively, when the command is executed. This makes it easy to set up dependencies between the commands. The Cmd class also has a member function registerInterface() that can be used to register a push button interface to the command. The Cmd class has a pure virtual function doit() that must be implemented by all the classes derived from it. The doit() member function is executed when the user activates

the button representing the command. Figure 18 illustrates the class inheritance hierarchy of the Cmd class.

4.3.7 CmdButtonInterface

The Cmd class described in Section 4.3.6 is an abstract class and is independent of any user interface component. The CmdButtonInterface class is derived from the Cmd class and provides a push button interface to the Cmd class, i.e., it uses push button widgets to represent commands. The Cmd class contains an array of CmdButtonInterface classes. Each time a new CmdButtonInterface object is created, it registers with the command class by calling the registerInterface() function in the Cmd class.

The CmdButtonInterface class also registers a callback to be called whenever the button is pressed. This callback function in turn calls the execute() member function in Cmd class, which calls the doit() function in the command object, to perform the desired action.

4.3.8 MenuBar

The MenuBar class is derived from the BaseWidget class and encapsulates the process of constructing a menubar with multiple pulldown menu panes. The MenuBar class has a member function addCommands() that can be used to add commands and to display them in the menu. The addCommands() member function takes a CmdList (a class containing a list of commands) and forms a pulldown menu from that. The MenuBar class is independent of the rest of the MotifApp framework and can be used whenever a menubar is needed [Young 92]. The MenuWindow automatically provides a menubar.

4.4 Limitations

The maximally parallel task system generating tool has the following limitations:

- The tool does not provide any means for detecting or removing cycles.
- Precedence graphs are shown only when the number of tasks in the task system is less than or equal to twenty. When the number of tasks become larger, the precedence graph may get cluttered.
- The adjacency matrices of the determinate task systems and the maximally parallel task systems can be viewed only when the number of tasks is less than or equal to thirty. When the number of tasks is greater than thirty, the window size becomes greater than the screen size. Providing a scrollbar may not be of great help since it is difficult to keep track of the rows and columns while scrolling.
- Arrowheads in the precedence graph were considered implicit, thus no arrowheads were provided to show dependencies. A top-down flow is assumed.
- Even though the random task generator can generate more than one task system at one time, the program runs by taking only one task system at a time as input.

CHAPTER V

EXPERIMENTATION

This chapter contains a brief description of the experiment conducted using the tool. In the sets of runs comprising the experimentation, the input task systems were obtained by using the random task system generator developed by Samadzadeh [Samadzadeh 92]. These task systems contained any number of tasks between 1 and 100. The number of memory cells were assumed to be 100, i.e., the memory cells were assumed to be from 1 to 100.

The program was run several times by keeping the size of the domains and ranges at 5 and varying the number of memory cells from 5 to 100 in steps of 5. Although these numbers were arbitrarily chosen, the idea was to cover as much of the spectrum of reasonable situations as possible. The program was run five times with each increment of the number of memory cells. In the average case analysis, the average of the five runs was considered. In the best case, the run for which the increase in degree of parallelism was maximum was considered, and in the worst case analysis, the run for which the increase was the least was considered.

Figures 19, 20, and 21 show the degree of parallelism for a determinate task system and the corresponding maximally parallel task system in the average case when the number of tasks are 10, 20, and 30, respectively. Figures 22 and 23 show the best and

worst case analysis respectively, when the number of tasks is 20. The graphs show that initially there is no increase in parallelism, but as the ratio of the number of memory cells to the number of domain and range cells increases, the degree of parallelism increases.

The number of tasks considered in describing the experiments (1 to 100) was arbitrary. Task Systems with more number of tasks were also tested but the results did not show any significant changes in the degree of parallelism. The increase in parallelism was generally found to be independent of the number of tasks but dependent on the ratio of the number of memory cells to the number of domain and range cells. As this ratio increased, the degree of parallelism was also found to increase, the best case being when all the tasks can be executed in parallel. Significant increase in parallelism was obtained when the ratio was greater than ten.

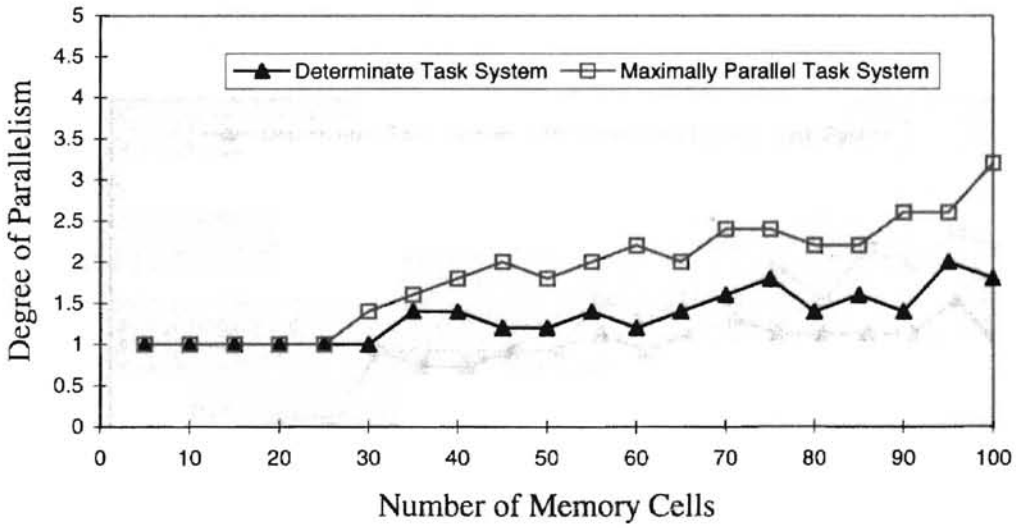


Figure 19. Average Case Analysis (number of tasks = 10 and the number of domain/range cells = 5)

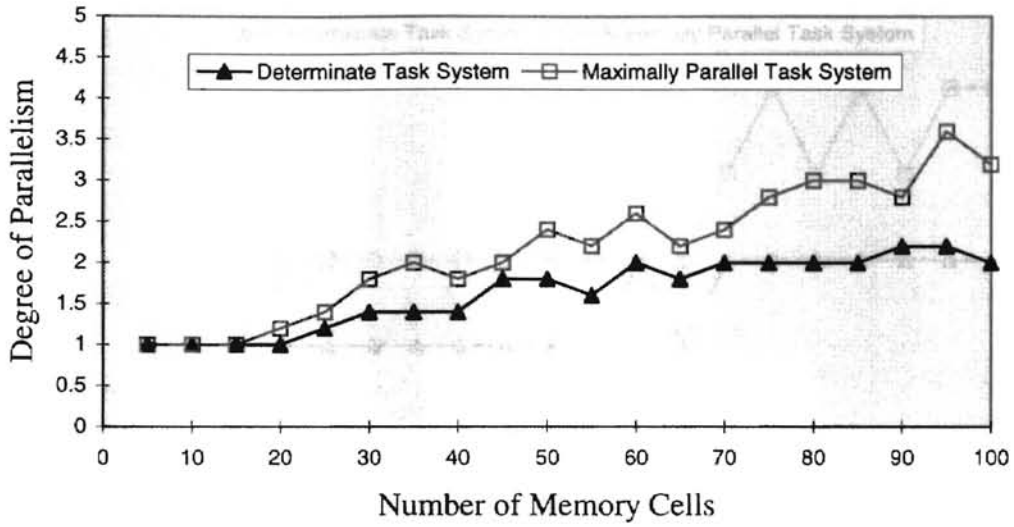


Figure 20. Average Case Analysis (number of tasks = 20 and the number of domain/range cells = 5)

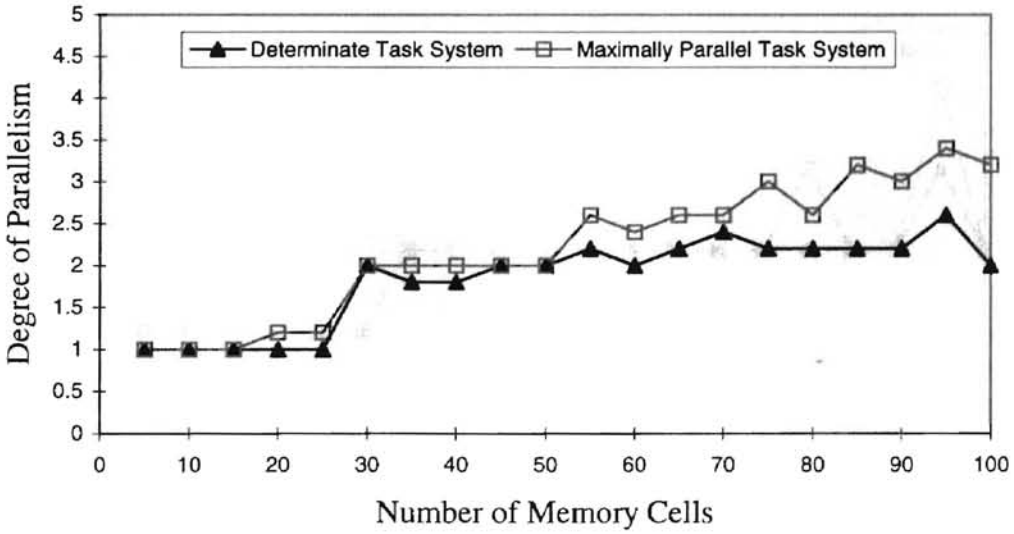


Figure 21. Average Case Analysis (number of tasks = 30 and the number of domain/range cells = 5)

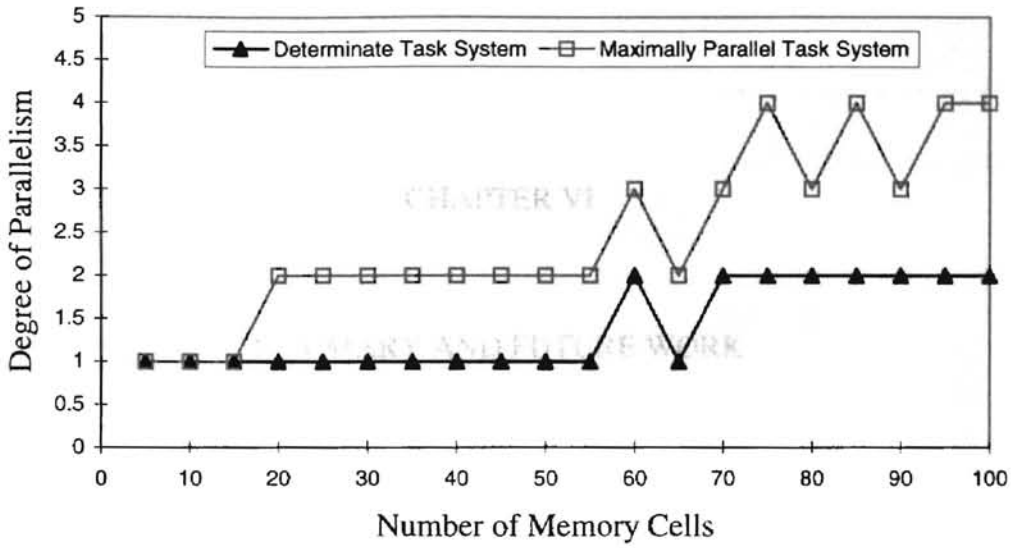


Figure 22. Best Case Analysis (number of tasks = 20 and the number of domain/range cells = 5)

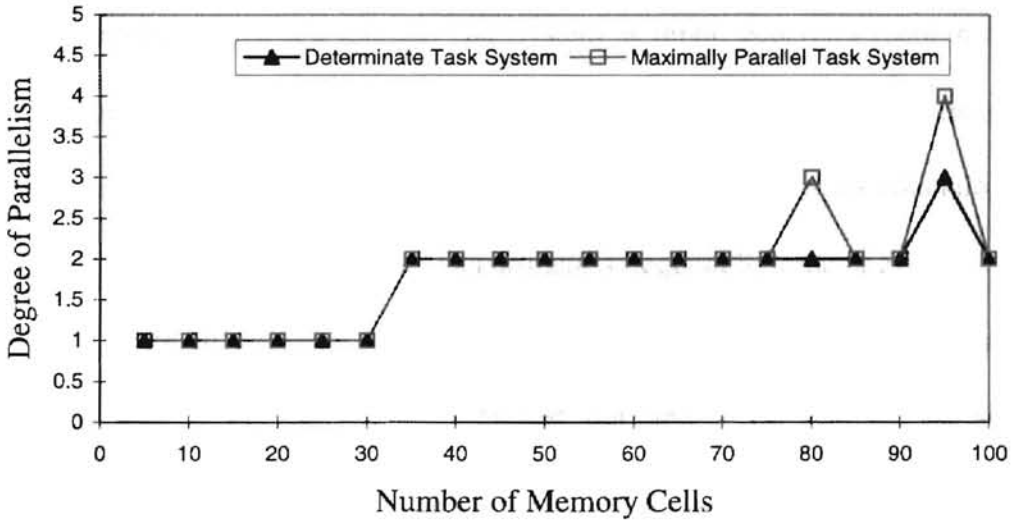


Figure 23. Worst Case Analysis (number of tasks = 20 and the number of domain/range cells = 5)

CHAPTER VI

SUMMARY AND FUTURE WORK

In Chapter I the main objective of the thesis was presented. Chapter II presented an introduction to task systems and the literature review for the topic. Chapter III contained a description of the implementation platform and the environment. In Chapter IV, the various design and implementation issues were discussed. Chapter V described the various tests conducted on the program, illustrated the results, and provided a brief summary of the evaluation of the tool.

The tool was designed and developed to aid in the study of maximal parallelism extraction algorithm. It takes a random task system as input, generates random domains and ranges for the tasks, makes it determinate, and then generates the corresponding maximally parallel task system. It was designed and developed as an educational tool running on the OSU Computer Science Department's Sequent Symmetry S/81.

Future work in this area could include the following. A thorough study on the trends of the graphs obtained can be carried out and a formula that predicts an approximate value for the degree of parallelism for specified values of the number of memory cells, the number of domain/ranges cells, and the number of tasks could be derived. Instead of considering a random task system, an actual program can be considered as the task system and the performance of the algorithm can be studied. A

criterion could be developed by which for a given task system, the tool uses an optimal determinate task system, using which the best possible increase in parallelism could be achieved. Since the program has limitations in displaying graphs when the number of tasks is greater than 20 and displaying adjacency matrices when the number of tasks is greater than 30, options can be provided to select a font using which the precedence graphs and adjacency matrices of task systems with a large number of tasks can be displayed. An option to resize and zoom the precedence graphs can be provided.

REFERENCES

[Baer 73] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," *ACM Computing Survey*, Vol. 5, No. 1, pp. 155-166, March 1973.

[Berlage 91] Thomas Berlage, *OSF/Motif: Concepts and Programming*, Addison Wesley Publishing Company, Reading, MA, 1991.

[Bernstein 66] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers*, Vol. EC-15, No. 5, pp. 757-763, October 1966.

[Bowman 95] Charles F. Bowman, *Objectifying Motif*, SIGS Books, New York, NY, 1995.

[Brain 92] Marshall Brain, *Motif Programming: The Essentials and More*, Butterworth-Heinemmann, Newton, MA, 1992.

[Coffman and Denning 73] Edward G. Coffman, Jr. and Peter J. Denning, *Operating Systems Theory*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[Desrochers 87] George R. Desrochers, *Principles of Parallel and Multiprocessing*, Intertext Publications, Inc., New York, NY, 1987.

[Fet 95] Yakov Fet, *Parallel Processing in Cellular Arrays*, John Wiley & Sons Inc., New York, NY, 1995.

[Hassan 93] M. T. Hassan, "Towards a Graphical Petri Net tool," Masters Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, 1993.

[Heller and Forguson 94] Dan Heller and Paula M. Forguson, *Motif Programming Manual*, O'Reilly & Associates, Inc., Sebastapol, CA, 1994.

[Nutt 92] Gary J. Nutt, *Centralized and Distributed Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[Nye 90] Andrian Nye, *X Protocol Reference Manual for Version 11 of the X Window System*, O'Reilly & Associates, Inc., Sebastapol, CA, 1990.

[Park and Miller 88] S. K. Park and K. W. Miller, "Random Number Generators: Good Ones Are Hard to Find", *Communications of the ACM*, Vol. 31, No. 10, October 1988, pp. 1192-1201.

[Samadzadeh 92] Farideh A. Samadzadeh, "Scheduling Algorithms for Parallel Execution of Computer Programs," Ph.D. Dissertation, Computer Science Department, Oklahoma State University, Stillwater, OK, 1992.

[Sequent 90] *DYNIX/ptx User's Guide,* Sequent Computer, Inc., 1990.

[Young 92] Douglas A. Young, *Object Oriented Programming with C++ and OSF/Motif*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

APPENDICES

APPENDIX A

GLOSSARY

Callback:	A callback is a mechanism to trigger a specific action in response to a user event.
Client-Server Model:	In a client-server model a process called server provides services to other processes called clients. In an X Window system, the X terminal acts as the server and the clients are the application programs.
DAG:	A DAG is a directed graph in which there exists no path that starts and ends at the same node.
Determinate Task System:	A task system is said to be determinate if it gives the same value sequence irrespective of the execution sequence.
Graphical User Interface (GUI):	A visual representation of a program's functionality that can be manipulated in a friendly and non-programmatic manner.
Motif Style Guide:	A set of guidelines and standards that helps develop consistency among Motif applications.
Mutually Non-Interfering Tasks:	Any two tasks in a task system are said to be mutually non-interfering if for any two tasks T and T' in the task system, one is either the successor or predecessor of the other, or if the following equation holds for the domains and ranges of the two tasks: $R_T \cap D_{T'} = R_T \cap R_{T'} = R_{T'} \cap D_T = \Phi$.
Parallel Processing:	Parallel processing denotes a wide range of techniques used to perform concurrent processing of

	tasks to increase the effective computational speed of a computer with multiple processors.
Task System:	A task system consists of a set of tasks with precedence among them.
Task:	A task is a unit of computation that is executed sequentially when assigned to a processor.
Widget:	A widget is a user interface component comprising of data structures and procedures that the user sees in the form of a picture such as a menu, a dialog box, etc.
Window Manager:	An X client that allows users to display and manipulate windows on the screen.
Window:	Windows are rectangular regions displayed on the screen.
X Protocol:	The X protocol controls the communication between the client and the server in the X Window System.
X Window System:	A hardware-independent and network transparent base layer that supports the development of Graphical User Interfaces.

APPENDIX B

TRADEMARK INFORMATION

DYNIX/ptx:	A registered trademark of Sequent Computer Systems, Inc.
NCD:	A registered trademark of Network Computing Devices, Inc.
OSF/Motif:	A Registered trademark of the Open Software Foundation.
Sequent Symmetry S/81:	A registered trademark of Sequent Computer Systems, Inc.
UNIX:	A registered trademark of AT&T.
X Window System:	A trademark of the Massachusetts Institute of Technology.

APPENDIX C

USER'S MANUAL

The following or the steps to be followed in using the tool:

1. At the UNIX prompt type the command /g/akamala/thesis/GUI/UMPTS.
2. A window as shown in Figure 13 appears on the screen.
3. In the File menu, select the 'Open' command to select an input file for the program to run on. In the file selection box that appears, select a fixed format input file (e.g., Fixed1.dat) or a random format input file (e.g., Random1.dat).
4. Select the 'Run' command in the File menu to run the algorithm.
5. In the View menu select the following commands. Select the 'Adjacency Matrix' command to view the adjacency matrices of the determinate task system and the corresponding maximally parallel task system, the 'Determinate Graph' command to view the precedence graph of the determinate task system, the 'Maximally Parallel Graph' command to view the precedence graph of the maximally parallel task system, and the 'Results' command to view the comparisons between the determinate task system and the maximally parallel task system.
6. Refer to the Help menu for more details about the program.

APPENDIX D

PROGRAM LISTINGS

```
////////////////////////////////////////////////////////////////////////////////
//                    Program : Generation of Maximally Parallel Task Systems
//                     Author : Kamalakar Ananthaneni
//                       Date : November 16
//        Programming Language : C++
//       Toolkits & Libraries : Motif, Xt Intrinsics, and Xlib
//
//  The user interface of the program was based on the MotifApp framework provided by
//  Douglas Young,Prentice Hall, 1992.
////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////
// BaseWidget.h: This is an abstract class serving as the base
//  for all other widgets. The BaseWidget class supports a _baseWidget
//  data member, that serves as the root of every component's widget
//  tree. The class also provides a baseWidget() member function that
//  provides public access to this widget. It also supports a _baseName
//  member that stores the name of the object.
//////////////////////////////////////////////////////////////////////
#ifndef PARENTWIDGET_H
#define PARENTWIDGET_H

#include <Xm/Xm.h>

class BaseWidget{

  protected:

    char    *_baseName;
    Widget   _baseWidget;

    // Protected constructor to prevent instantiation.
    BaseWidget(const char * );

  public:

    virtual void manage();     // Manage all the derived widgets.
                               // When a widget is managed, it becomes visible.
    virtual void unmanage();   // This function destroys the widget tree for which,
                               // _baseWidget is the parent widget.

    // Function that handles the destruction of a widget.
    void handleDestruction();

    // Callback function to be called when the Widget is destroyed.
    static void destroyedCallback( Widget, XtPointer, XtPointer);
    virtual void widgetDestroyed();

    virtual ~BaseWidget();
    const Widget baseWidget();
};
#endif
```

```
//////////////////////////////////////////////////////////
// BaseWidget.C: Initial version of a class to define
//  a protocol for all components.
//////////////////////////////////////////////////////////
#include "BaseWidget.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>

// Constructor for the BaseWidget class.
BaseWidget::BaseWidget (const char *name )
{
    _baseWidget = NULL;
    assert ( name != NULL );  // Make sure programmer provides a name
                              // to all widgets.
    _baseName = new char[strlen(name)+1];
    strcpy(_baseName, name);
}

// Destructor for the BaseWidget class.
BaseWidget::~BaseWidget()
{
    // Check if the Widget is not already destroyed.
    // If it still exists destroy it.
    if( _baseWidget )
        XtDestroyWidget ( _baseWidget );
    delete _baseName;
}

// The manage() function calls XtManageChild() to manage the root of the
// widget tree.
void BaseWidget::manage()
{
    // Use the assert macro to ensure that XtManageChild() is not called
    // with a NULL widget and to make sure that the derived classes create
    // a base widget.
    assert ( _baseWidget != NULL );
    XtManageChild ( _baseWidget );
}

// The unmanage() function call XtUnmanageChild() to unmanage the root
// of the widget tree.
void BaseWidget::unmanage()
{
    // Make sure that a NULL widget is not being unmanaged.
    assert ( _baseWidget != NULL );
    XtDestroyWidget( _baseWidget);
}

// This function returns an instance of the base widget. It can be used
// as a parent for other widgets.
const Widget BaseWidget::baseWidget()
{
 return _baseWidget;
}

// This function is called from the destroyedCallback() function when
// the widget is destroyed.
void BaseWidget::widgetDestroyed()
{
 _baseWidget = NULL;        // Assign NULL to avoid dangling pointer references.
}

// This function handles the destruction of the widget by registering
// a callback function to be called when the widget is destroyed.
void BaseWidget::handleDestruction()
{
 assert( _baseWidget != NULL);
 XtAddCallback(_baseWidget,
               XmNdestroyCallback,
               &BaseWidget::destroyedCallback,
               (XtPointer) this );
}
```

```
// This is the callback function called when the widget gets destroyed.
void BaseWidget::destroyedCallback(Widget, XtPointer clientData, XtPointer)
{
 // Cast the clientData to the expected object type.
 BaseWidget * obj = ( BaseWidget *) clientData;


// Call the corresponding member function.
 obj->widgetDestroyed();
}

////////////////////////////////////////////////////////////////////////////////////////
// Application.h:
// The Application class captures characteristics common to most Motif applications.
// It provides a place to store data that may be needed throughout an application.
// It contains a pointer to the X Display structure, the application context required
// by many Xt functions, the name of the application and the class name of the application
// as data members. The protected part of the class contains member functions initialize()
// and handleEvents(). These functions are called from the main(), which is a friend of
// the application class. The private portion of the class has two member functions for
// registering and unregistering MainWindow objects. The MainWindow class is declared
// as a friend of the Application class.
// The Application header file exports a pointer to an instance of the
// Application class, theApplication, which is available to any class that
// includes the Application.h header file. Each application must create a
// single instance of the Application class and can be accessed through
// out the program as theApplication.
////////////////////////////////////////////////////////////////////////////////////////
#ifndef APPLICATION_H
#define APPLICATION_H
#include "BaseWidget.h"

class Application : public BaseWidget{

    // Allow main and MainWindow to access protected and private member functions of
    // Application class, by making them friend functions of Application class.

    friend void main (unsigned int, char ** ); // main needs to call the initialize() and
                                               // handleEvents() functions.

    friend class MainWindow;  // MainWindow needs to call private member functions
                              // for registering windows with the application.
  private:

    // Functions for registering and unregistering toplevel windows.

    void registerWindow ( MainWindow * );
    void unregisterWindow ( MainWindow * );

  protected:

    // Support commonly needed data structures as a convenience.

    Display     *_display;    // Pointer to the X Display structure.
    XtAppContext _appContext; // Application Context.

    // Functions to handle Xt interface.
    virtual void initialize ( unsigned int *, char ** );

    // Handle events by entering an event loop.
    virtual void handleEvents();

    char   *_applicationClass;    // Class name of this application.
    MainWindow  **_windows;       // The top-level window in the program.
    int      _numWindows;         // Number of toplevel windows in the program.

  public:

    Application ( char * );
    virtual ~Application();

    // Functions to manipulate application's top-level windows.

    // Manage all the registered windows.
    void manage();
```

```
    // Unmanage all the registered windows.
    void unmanage();

    // Iconify all the registered windows.
    void iconify();
    // Convenient access functions.
    Display      *display()      { return _display; }
    XtAppContext  appContext()   { return _appContext; }
    const char   *applicationClass()  { return _applicationClass; }

    virtual const char *const className() { return "Application"; }
};

// Pointer to single global instance. This global instance is instantiated and used to
// provide the base widget for all top-level widgets.
extern Application *theApplication;

#endif

////////////////////////////////////////////////////////////////
// Application.C:  This file contains all the member functions of
// the Application class.
////////////////////////////////////////////////////////////////
#include "Application.h"
#include "MainWindow.h"
#include <assert.h>
#include <stdlib.h>
#include <iostream.h>

Application *theApplication = NULL;

// The Application class's constructor.
Application::Application ( char *appClassName ) : BaseWidget(appClassName )
{
    // Set the global Application pointer, so that it can be used as parent for all
    // top-level widgets.

    theApplication = this;

    // Initialize data members

    _display    = NULL;
    _appContext = NULL;
    _windows    = NULL;
    _numWindows = 0;
    _applicationClass = strdup ( appClassName );
}

// The initialize function initializes the Xt Intrinsics and creates
// the Application object's base widget.
void Application::initialize ( unsigned int *argcp, char **argv )
{
    // The XtAppInitialize function, initializes the Xt Intrinsics,
    // creates an application context and opens a connection to the
    // X server. It creates the toplevel shell widget used as a parent
    // for all other widget.
    _baseWidget = XtAppInitialize ( &_appContext,
                                    _applicationClass, NULL, 0,
                                    argcp, argv,
                                    NULL, NULL, 0 );

    // Extract and save a pointer to the X display structure.
    _display = XtDisplay ( _baseWidget );

    // Call the function in BaseWidget to handle destruction.
    handleDestruction();

    // Center the shell and make sure it isn't visible. The shell is made
    // invisible so that when a widget is created as it's child, only that
    // widget and not both the shell and the widget are displayed.
    XtVaSetValues ( _baseWidget,
                    XmNmappedWhenManaged, False,
                    XmNx, DisplayWidth ( _display, 0 ) / 2,
                    XmNy, (DisplayHeight ( _display, 0 ) * 2) / 3,
                    XmNwidth,  1,
                    XmNheight, 23,
```

```
                NULL );

    // The instance name of this object was set in the BaseWidget
    // constructor before the name of the program was available, because
    // this widget is used as the base widget of the application.
    // Free the old name and reset it to argv[0].

    delete _baseName;
    _baseName = strdup ( argv[0] );

    // Force the shell window to exist so dialogs popped up from
    // this shell behave correctly.

    XtRealizeWidget ( _baseWidget);

    // Initialize and manage any windows registered
    // with this application.
    for ( int i = 0; i < _numWindows; i++ )
    {
        _windows[i]->initialize();
        _windows[i]->manage();
    }
}

// Destructor for the Application class.
Application::~Application()
{
    // Free the dynamically allocated memory.
    delete _applicationClass;
    delete _windows;
}

// This function handles events by entering the event loop.
void Application::handleEvents()
{
    // Just loop forever
    XtAppMainLoop ( _appContext );
}

// This function is called from the MainWindow class to register
// a main window with the application.
void Application::registerWindow ( MainWindow *window )
{
    int i;
    MainWindow **newList;

    // Allocate a new list large enough to hold the new
    // object and copy the contents of the current list
    // to the new list.

    newList = new MainWindow*[_numWindows + 1];

    for ( i = 0; i < _numWindows; i++ )
    newList[i] = _windows[i];

    // Install the new list and add the main window to be registered to the list.

    delete []_windows;
    _windows =  newList;
    _windows[_numWindows] = window;

    _numWindows++;
}

// This function is called from the MainWindow class to unregister
// a main window. The window passed as argument is removed from the array
// _windows.
void Application::unregisterWindow ( MainWindow *window )
{
        int i, index;
    MainWindow **newList;

    // Allocate a new smaller list of windows, since one of the registered windows needs
    // to be removed from the list of registered windows.

    newList = new MainWindow*[_numWindows - 1];
```

```
    // Copy all objects, except the one to be
    // removed, to the new list of registered windows.

    index = 0;

    for ( i = 0; i < _numWindows; i++ )
        if ( _windows[i] != window )
            newList[index++] = _windows[i];


    // Install the new list
    delete []_windows;
    _windows =  newList;

    _numWindows--;
}

// This function manages all registered windows.
void Application::manage()
{
    // Manage all application windows. This will pop up
    // iconified windows as well.

    for ( int i = 0; i < _numWindows; i++ )
        _windows[i]->manage();
}

// This function unmanages all registered windows by calling the
// unmanage() function in the MainWindow class.
void Application::unmanage()
{
    // Unmanage all application windows.

    for ( int i = 0; i < _numWindows; i++ )
        _windows[i]->unmanage();
}

// This function iconifies all registered windows by calling the
// iconify() function in the MainWindow class.
void Application::iconify()
{

 // Iconify all the application windows.
 for(int i=0; i < _numWindows; i++)
 _windows[i]->iconify();
}

//////////////////////////////////////////////////////////////////////
// Main.C: The main() function, initializes the application
// by calling the initialize() member function in the Application
// class. theApplication is a global instance of the application
// class. Then it calls the handleEvents() function in the
// application class to handle events by entering the event loop.
//////////////////////////////////////////////////////////////////////
#include "Application.h"
#include <assert.h>
#include <iostream.h>


void main (unsigned int argc, char **argv )
{
    // Make sure an Application object exists, since it is used as the base widget
    // of all top-level windows, to initialize the application, and to handle the
    // events.
    assert ( theApplication );

    // Initialize the Xt Intrinsics, build all windows, and enter event loop.
    theApplication->initialize ( &argc, argv );
    // Handle the events by entering an event loop.
    theApplication->handleEvents();
}

//////////////////////////////////////////////////////////////////////
// BaseWindow.h: This BaseWindow is used as the base class for the
//  MatrixWindow and the ResultsWindow classes.
//////////////////////////////////////////////////////////////////////
```

```
#ifndef BASEWINDOW_H
#define BASEWINDOW_H
#include "BaseWidget.h"
class BaseWindow : public BaseWidget
{
 protected:

    Widget _baseWindow;
    char* _windowName;

 public:
    BaseWindow(const char*);
    virtual void manage();      // Displays the window by realizing it.
    virtual void unmanage();    // Destroys the window and its children.

    const Widget baseWindow(); // Returns _baseWindow.

    void initialize();          // Initialize the window.

    virtual void createShell()=0;       // Pure virtual functions to be implemented
    virtual void createBaseWidget()=0; // by the derived classes.

    virtual ~BaseWindow();
};
#endif


#include "BaseWindow.h"
#include <string.h>
#include <iostream.h>
#include <assert.h>

// Constructor.
BaseWindow::BaseWindow(const char* name):BaseWidget(name)
{
    _baseWindow = NULL;

    if(name)
    {
      _windowName = new char[strlen(name)+1];
      strcpy(_windowName, name); // Set the name of the window.
     }
}

// Display the window by realizing it and popping it up.
void BaseWindow::manage()
{
    assert(_baseWindow);
    // If the window is not realized, realize the window by calling the XtRealizeWidget
    // function.
    if(!XtIsRealized(_baseWindow))
         XtRealizeWidget(_baseWindow);

    XtPopup(_baseWindow, XtGrabNone); // Popup the window.
    XMapRaised(XtDisplay(_baseWindow),XtWindow(_baseWindow));
}

// Destroy the widget.
void BaseWindow::unmanage()
{
     BaseWidget::unmanage();
     assert(_baseWindow); // Check if the _baseWindow is not NULL.
     XtDestroyWidget(_baseWindow);
}

const Widget BaseWindow::baseWindow()
{
    return _baseWindow;
}

// Create the shell widget and the base widget by calling the CreateShell() and
// CreateBaseWidget() member functions in the derived class. The base widget is used as
// the parent widget for the derived classes.
void BaseWindow::initialize()
{
    createShell(); //Call the derived classes to create the shell
```

```
    XtVaSetValues(_baseWindow,
              XmNtitle, _windowName,
              XmNdeleteResponse, XmUNMAP,
              NULL);

    // Call the derived class's createWidget function to create the
    // base widget.
    createBaseWidget();

    // Manage the window and its children.
    if(!XtIsManaged(_baseWidget))
     XtManageChild(_baseWidget);
}

BaseWindow::~BaseWindow()
{
    assert(_baseWindow);          // Make sure _baseWindow is not null before
    XtDestroyWidget(_baseWindow); // destroying it.
    XtFree(_windowName);
}

////////////////////////////////////////////////////////////////////////////
// MatrixWindow.h: This class contains the widgets needed to display the adjacency
//  matrix of the determinate and the maximally parallel task systems.
//  The MatrixWindow class is derived from the BaseWindow class and contains a
//  label widget and a push button widget.
//  The label widget displays the adjacency matrix and the push button widget
//  is used to pop down the widget when the user is done viewing the matrix.
////////////////////////////////////////////////////////////////////////////
#ifndef MATRIXWINDOW_H
#define MATRIXWINDOW_H

#include "BaseWindow.h"
#include "UMPTSApp.h"
#define MAXPLL 1
#define DETERMINATE 0

class MatrixWindow: public BaseWindow
{
 private:

  // Private data members.
    Widget _okButton;
    Widget _matrixLabel;

    // Function to be called when the OK button is activated.
    static void okCallback(Widget, XtPointer, XtPointer);

 public:
    MatrixWindow(char*);
    void okExecute();

    // Called from the BaseWindow::initialize() member function to create
    // the pop up shell.
    virtual void createShell();

    // Called from the BaseWindow::initialize() function to create the
    // base widget for the window.
    virtual void createBaseWidget();

    void manage();

    // The initialize() member function calls the initialize() member function
    // in the base window to create the shell and the base widgets.
    int initialize();

    // This function places the adjacency matrix in the window.
    void showMatrix(int);
    ~MatrixWindow(void);
};
#endif

////////////////////////////////////////////////////////////////////////
// MatrixWindow.C
////////////////////////////////////////////////////////////////////////
#include "MatrixWindow.h"
```

```
#include "Application.h"
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/Label.h>
#include <string.h>
#include <stdio.h>
#include "UMPTSApp.h"
#include "DialogManager.h"

// Constructor
MatrixWindow::MatrixWindow(char* Name):BaseWindow(Name)
{
  //EMPTY
}

// This member function calls the initialize() member function in the
// the BaseWindow class to create the shell widget and to create the
// base widget. This function sets the width and height of the base
// window based on the number of tasks.

int MatrixWindow::initialize()
{
   int Width = theUMPTSApp->_determinate->_numOfTasks*21+100;
   int Height = (theUMPTSApp->_determinate->_numOfTasks+3)*21;
   char str[100],str1[50];

   //Create the topshell and the form widget by calling the baseWidget's
   //initialize member function
   BaseWindow::initialize();

   // If the size of the window required to display the matrix is bigger
   // than the size of the screen, display a message saying that the matrix
   // cannot be viewed.
   if((Width > 999) || (Height > 725))
   {
       strcpy(str, "Cannot view adjacency matrix.\n");
       strcpy(str1,"Window size too big to fit in screen.");
       strcat(str,str1);
       theInfoDialogManager->post(str,(void*)this);
       return 1;
   }

    // Set the height and width of the window.
    XtVaSetValues(_baseWindow,
                  XmNwidth, Width,
                  XmNheight, Height,
                  XmNminWidth, 300,
                  XmNminHeight, Height,
                  XmNmaxWidth, Width,
                  XmNmaxHeight, Height,
                  NULL);

    // Create the label and the push button widgets.
    _matrixLabel = XtVaCreateWidget("Matrix",
                                    xmLabelWidgetClass, _baseWidget,
                                    NULL);

    _okButton = XtVaCreateWidget("OK",
                                 xmPushButtonWidgetClass, _baseWidget,
                                 NULL);

    // Add the callback function to be invoked when the OK button is invoked.
    XtAddCallback(_okButton,
                  XmNactivateCallback, &MatrixWindow::okCallback,
                  (XtPointer)this);

    // Attach the label to the form so that, it's bottom widget is the
    // push button widget.
    XtVaSetValues(_matrixLabel,
                  XmNtopAttachment, XmATTACH_FORM,
                  XmNleftAttachment, XmATTACH_FORM,
                  XmNrightAttachment, XmATTACH_FORM,
                  XmNbottomAttachment, XmATTACH_WIDGET,
                  XmNbottomWidget, _okButton,
                  NULL);
```

```
    XtVaSetValues(_okButton,
                  XmNrightAttachment, XmATTACH_FORM,
                  XmNbottomAttachment, XmATTACH_FORM,
                  XmNleftAttachment, XmATTACH_FORM,
                  NULL);
    return 0;
}

// This function creates a popup shell. This function is called from
// the initialize() function in the BaseWindow class.
void MatrixWindow::createShell()
{
    // Create a popop shell widget, used as the toplevel window.
    _baseWindow = XtVaCreatePopupShell(_windowName,
                                       topLevelShellWidgetClass,
                                       theApplication->baseWidget(),
                                       XmNtitle, "Adjacency Matrix",
                                       NULL);

}

// This function is called from the initialize() function in the BaseWindow
// class and creates the base widget for the window.
void MatrixWindow::createBaseWidget()
{
   // Create a form widget as the base widget.
    _baseWidget = XtVaCreateWidget(_baseName,
                                   xmFormWidgetClass, _baseWindow,
                                   NULL);
}

// Manage the window and all it's children.
void MatrixWindow::manage()
{
    BaseWindow::manage();

    XtManageChild(_okButton);
    XtManageChild(_matrixLabel);
}

// This function displays the adjacency matrix. The Flag indicates if
// the matrix to be displayed is the determinate matrix or the maximally
// parallel matrix. The matrix is displayed as a character string.
void MatrixWindow::showMatrix(int Flag)
{

     char temp[500];
     char *string;

     char *nameString = NULL;
     XmStringCharSet char_set = XmSTRING_DEFAULT_CHARSET;
     XmString Str;

     XFontStruct *font = NULL;
     XmFontList fontList = NULL;
     int strSize; // Size of the string to be displayed.

     // Calculate the number of characters needed to display the matrix.
     // This includes the spaces between columns and the rows.
     strSize = theUMPTSApp->_determinate->_numOfTasks *
                  theUMPTSApp->_determinate->_numOfTasks * 4;

     // Allocate memory for the string.
     string = new char[strSize];

     // Select a suitable font to display the matrix.
     nameString = "-ncd*medium*11*";
     font = XLoadQueryFont(XtDisplay(_matrixLabel),nameString);
     fontList = XmFontListCreate(font, char_set);

     sprintf(string, "\n    ");
     for(int i=0;i< theUMPTSApp->_determinate->_numOfTasks;i++)
    {
         sprintf(temp, "%d ",i%10);
         strcat(string, temp);
    }
```

```
    strcat(string, "\n\n");

    switch(Flag)
    {

      // The windows are positioned in such a way that the window
      // showing the adjacent matrix of the determinate task system,
      // does not completely overlap the window showing the determinate
      // task system.

       case DETERMINATE:

               for(i=0;i<theUMPTSApp->_determinate->_numOfTasks;i++)
               {
                   // Display the task number before each row for clarity.
                   sprintf(temp,"%2d: ",i);
                   strcat(string, temp);

                   // Add the matrix to the string to be displayed.
                   for(int j=0;j<theUMPTSApp->_determinate->_numOfTasks;j++)
                   {
                      sprintf(temp, "%c ",theUMPTSApp->_determinate->_adjacency[i][j]);
                      strcat(string, temp);
                   }
                   strcat(string, "\n");
               }

               XtVaSetValues(_baseWindow,
                             XmNx,200,
                             XmNy,100,
                             NULL);

               break;

      case MAXPLL:

               for(i=0;i<theUMPTSApp->_maximallyPll->_numOfTasks;i++)
               {
                   // Display the task number before each row for clarity.
                   sprintf(temp,"%2d: ",i);
                   strcat(string, temp);

                   // Add the adjacency matrix to the string to be displayed.
                   for(int j=0;j<theUMPTSApp->_maximallyPll->_numOfTasks;j++)
                   {
                      sprintf(temp, "%c ",theUMPTSApp->_maximallyPll->_adjacency[i][j]);
                      strcat(string, temp);
                   }
                   strcat(string, "\n");
               }

               XtVaSetValues(_baseWindow,
                             XmNx,350,
                             XmNy,200,
                             NULL);
               break;

      default:   break;
    }

    // Make an XmString from the character string by using the function
    // XmStringCreateLtoR.
    Str = XmStringCreateLtoR(string ,XmSTRING_DEFAULT_CHARSET);

    XtVaSetValues(_matrixLabel,
               XmNlabelString, Str,
               XmNalignment, XmALIGNMENT_BEGINNING,
               XmNfontList, fontList,
               NULL);

   XmStringFree(Str);

}

// This is the callback function called when the OK button is pressed.
```

```
void MatrixWindow::okCallback(Widget, XtPointer clientData, XtPointer)
{
    MatrixWindow* obj = (MatrixWindow*) clientData;
    obj->okExecute();
}


// Unmanage the window when the OK button is pressed.
void MatrixWindow::okExecute()
{  // When the window is unmanaged, it is no longer visible.
    BaseWindow::unmanage();
}


// Destructor for the class.
MatrixWindow::~MatrixWindow()
{
}

//////////////////////////////////////////////////////////////////
// ResultsWindow.h : Displays the results
//////////////////////////////////////////////////////////////////
#ifndef RESULTSWINDOW_H
#define RESULTSWINDOW_H

#include "BaseWindow.h"
#include "UMPTSApp.h"
class ResultsWindow: public BaseWindow
{
 private:

    Widget _okButton;
    Widget _resultsLabel; // Label widget to display the results.

    // Callback function to be called when the user selects the OK button.
    static void okCallback(Widget, XtPointer, XtPointer);

 public:

    ResultsWindow(void);
    void okExecute();

    // Override the pure virtual functions createShell() and createBaseWidget()
    // in the BaseWindow class.
    virtual void createShell();
    virtual void createBaseWidget();

    void manage();      // Manage the window.
    void initialize(); // Initialize the window and set the resources.

    void setResults(); // Display the results.
    ~ResultsWindow(void);
};
#endif

//////////////////////////////////////////////////////////////////////////
// ResultsWindow.C: Window that displays the comparisons between the
// determinate and the maximally parallel task systems.
//////////////////////////////////////////////////////////////////////////
#include "ResultsWindow.h"
#include "Application.h"
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/Label.h>
#include <string.h>
#include <stdio.h>
#include "UMPTSApp.h"

// Constructor.
ResultsWindow::ResultsWindow(void):BaseWindow("Results")
{
//EMPTY
}


// Initialize the window, by creating the widgets and setting the resources.
```

```
void ResultsWindow::initialize()
{

    //Create the topshell and the form widget by calling the baseWidget's
    //initialize member function
    BaseWindow::initialize();


    // Create the label and the pushButton widgets.
    _resultsLabel = XtVaCreateWidget("Results",
                                     xmLabelWidgetClass, _baseWidget,
                                     NULL);
    _okButton = XtVaCreateWidget("OK",
                                 xmPushButtonWidgetClass, _baseWidget,
                                 NULL);

    XtAddCallback(_okButton,
                  XmNactivateCallback, &ResultsWindow::okCallback,
                  (XtPointer)this);

    // Set the resources such that the OK button comes at the bottom of the label.
    XtVaSetValues(_resultsLabel,
                  XmNtopAttachment, XmATTACH_FORM,
                  XmNleftAttachment, XmATTACH_FORM,
                  XmNrightAttachment, XmATTACH_FORM,
                  XmNbottomAttachment, XmATTACH_WIDGET,
                  XmNbottomWidget, _okButton,
                  NULL);

    XtVaSetValues(_okButton,
                  XmNrightAttachment, XmATTACH_FORM,
                  XmNbottomAttachment, XmATTACH_FORM,
                  XmNleftAttachment, XmATTACH_FORM,
                  NULL);
}

// Overrides the pure virtual function createShell in the BaseWindow class.
// This function is called from the initialize() member function in the
// BaseWindow class.
void ResultsWindow::createShell()
{
   _baseWindow = XtVaCreatePopupShell(_windowName,
                                      topLevelShellWidgetClass,
                                      theApplication->baseWidget(),
                                      XmNtitle, _windowName,
                                      XmNwidth,400,
                                      XmNheight,270,
                                      XmNminWidth, 400,
                                      XmNminHeight, 270,
                                      XmNmaxWidth, 400,
                                      XmNmaxHeight, 270,
                                      XmNx, 400,
                                      XmNy,100,
                                      NULL);
}

// Overrides the pure virtual function in the BaseWindow class.
// This function is called from the BaseWindiw::initialize() function to
// create the base widget.
void ResultsWindow::createBaseWidget()
{
   _baseWidget = XtVaCreateWidget(_baseName,
   xmFormWidgetClass, _baseWindow,
   NULL);
}

// Manage the window and it's children.
void ResultsWindow::manage()
{
   BaseWindow::manage();

   XtManageChild(_okButton);
   XtManageChild(_resultsLabel);
}

// Displays the results.
```

```
void ResultsWindow::setResults()
{
   // Here set the string to be displayed
   // in the results window

   char temp[100];
   char string[1000];

   XmString Str;
   sprintf(temp,"                          Determinate Task System       UMPTS\n");
   strcpy(string, temp);
   sprintf(temp,"                          ----------------------        -----\n");
   strcat(string, temp);
   sprintf(temp, "Degree of Parallelism: %13d %20d\n\n",
           theUMPTSApp->_determinate->_maxDegOfParallelism,
           theUMPTSApp->_maximallyPll->_maxDegOfParallelism);
   strcat(string, temp);
   sprintf(temp, "Number of Dependencies: %12d %20d\n\n",
            theUMPTSApp->_determinate->_numRdntDependencies,
            theUMPTSApp->_maximallyPll->_numRdntDependencies);
   strcat(string, temp);

   Str = XmStringCreateLtoR(string, XmSTRING_DEFAULT_CHARSET);

   XtVaSetValues(_resultsLabel,
                 XmNlabelString, Str,
                 XmNalignment, XmALIGNMENT_BEGINNING,
                 NULL);

   XmStringFree(Str);
}

// Callback called when the OK button is selected.
// The window is unmanaged in response to the activation of the OK button.
void ResultsWindow::okCallback(Widget, XtPointer ClientData, XtPointer)
{
   ResultsWindow* obj = (ResultsWindow*) clientData;
   obj->okExecute();
}

void ResultsWindow::okExecute()
{
   BaseWindow::unmanage();
}

// Destructor.
ResultsWindow::~ResultsWindow()
{
}

//////////////////////////////////////////////////////////////////////
// MainWindow.h: Support a toplevel window
//////////////////////////////////////////////////////////////////////
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include "BaseWidget.h"

class MainWindow : public BaseWidget {

  protected:

   Widget   _main;         // The XmMainWindow widget.

  public:

    MainWindow ( char * );   // Constructor requires only a name
    virtual ~MainWindow();

    // The Application class automatically calls initialize()
    // for all registered main window objects.
    virtual void initialize();

    virtual void manage();   // Popup the window.
    virtual void unmanage(); // Pop down the window.
    virtual void iconify();  // Iconify the window.
};
```

```
extern MainWindow *theMainWindow;
#endif;

////////////////////////////////////////////////////////////////////////
// MainWindow.C: Supports a toplevel window.
////////////////////////////////////////////////////////////////////////
#include "Application.h"
#include "MainWindow.h"
#include <Xm/MainW.h>
#include <assert.h>
#include <iostream.h>

MainWindow::MainWindow ( char *name ) : BaseWidget( name )
{
    assert ( theApplication ); // Application object must exist
                               // before any MainWindow object is created.
    theApplication->registerWindow ( this );
}

void MainWindow::initialize( )
{

    // All toplevel windows in the MotifApp framework are
    // implemented as a popup shell off the Application's
    // base widget.
    _baseWidget = XtVaCreatePopupShell ( _baseName,
                                         applicationShellWidgetClass,
                                         theApplication->baseWidget(),
                                         XmNiconX, 300,
                                         XmNiconY, 300,
                                         0 );

// Call the handle destruction function in the BaseWidget class
handleDestruction();

_main = XtVaCreateManagedWidget ( "mainWindow",
                                  xmMainWindowWidgetClass, _baseWidget,
                                  XmNscrollingPolicy, XmAUTOMATIC,
                                  XmNscrollBarDisplayPolicy, XmAS_NEEDED,
                                  NULL);

}

MainWindow::~MainWindow( )
{
    // Unregisters this window with the Application object.
    theApplication->unregisterWindow ( this );
}

void MainWindow::manage()
{
    assert ( _baseWidget );
    XtPopup ( _baseWidget, XtGrabNone );

    // Map the window, in case the window is iconified.

    if ( XtIsRealized ( _baseWidget ) )
        XMapRaised ( XtDisplay ( _baseWidget ), XtWindow ( _baseWidget ) );
}

void MainWindow::unmanage()
{
    assert ( _baseWidget
    BaseWidget::unmanage();
}

void MainWindow::iconify()
{
    assert(_baseWidget);

    // Set the window to have an initial iconic state
    // in case the base widget has not yet been realized

    XtVaSetValues (_baseWidget, XmNiconic, TRUE, NULL);
```

```
    // If the widget has already been realized,
    // iconify the window
     if(XtIsRealized (_baseWidget))
     XIconifyWindow(XtDisplay(_baseWidget), XtWindow(_baseWidget), 0);
}

////////////////////////////////////////////////////////////
// MenuBar.h: A menu bar, whose panes support items
//  that execute commands.
////////////////////////////////////////////////////////////
#ifndef MENUBAR_H
#define MENUBAR_H
#include "BaseWidget.h"

class Cmd;
class CmdList;

class MenuBar : public BaseWidget{

  public:

    MenuBar ( Widget, char * );

    // Create a named menu pane from a list of Cmd objects.
    virtual void addCommands ( CmdList *, char * );

    virtual const char *const className() { return "MenuBar"; }
};
#endif

////////////////////////////////////////////////////////////
// MenuBar.C: A menu bar whose panes support items
//  that execute commands.
////////////////////////////////////////////////////////////
#include "MenuBar.h"
#include "Cmd.h"
#include "CmdList.h"
#include "CmdButtonInterface.h"
#include <Xm/RowColumn.h>
#include <Xm/CascadeB.h>

// Constructor.
MenuBar::MenuBar ( Widget parent, char *name ) : BaseWidget( name )
{
    // Base widget is a Motif menu bar widget

    _baseWidget = XmCreateMenuBar ( parent, _baseName, NULL, 0 );

    // Call the handleDestruction() function in the BaseWidget class for freeing
    // memory and avoiding dangling pointers.
    handleDestruction();
}

// This function adds the commands in the list to the MenuBar.
void MenuBar::addCommands ( CmdList *list, char *name )
{
    int    i;
    Widget pulldown, cascade;

    // Pointer to a font structure. Used to set the fonts.
    XFontStruct *font=NULL;
    XmFontList fontList=NULL;
    char *nameString=NULL;

    XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

    // Creates a pulldown menu pane.
    pulldown = XmCreatePulldownMenu ( _baseWidget, name, NULL, 0 );

    // Each entry in the menu bar must have a cascade button
    // from which the user can pull down the pane.
    cascade = XtVaCreateWidget ( name,
                                 xmCascadeButtonWidgetClass,
                                 _baseWidget,
                                 XmNsubMenuId, pulldown,
                                 NULL );
```

```
    XtManageChild ( cascade ); // Manage the cascade button.

    // Select the needed font for the menu items.
    nameString = "lucidasans-bold-10";
    font = XLoadQueryFont(XtDisplay(pulldown),nameString);
    fontList = XmFontListCreate(font, char_set);
    XtVaSetValues(cascade,
                  XmNfontList, fontList,
                  NULL);

    // Loop through the cmdList, creating a menu
    // entry for each command.

    for ( i = 0; i < list->size(); i++)
    {
        CmdButtonInterface *ci;                    // Create a push button interface for
                                                   // each command.
        ci  = new CmdButtonInterface ( pulldown, (*list)[i] );
        ci->manage();
    }
}

////////////////////////////////////////////////////////////
// MenuWindow.h: The MenuWindow class provides a menubar and the
//  commands in the menu panes.
////////////////////////////////////////////////////////////

#ifndef MENUWINDOW_H
#define MENUWINDOW_H

#include "MainWindow.h"

class MenuBar;
class Cmd;

class MenuWindow : public MainWindow {

  protected:

    MenuBar *_menuBar;

    // Maintain pointers to Cmd objects used throughout the application.

    Cmd *_exit;
    Cmd *_open;
    Cmd *_run;
    Cmd *_detGraph;
    Cmd *_maxPllGraph;
    Cmd *_adjacencyMatrix;
    Cmd *_results;
    Cmd *_howToRun;
    Cmd *_aboutUMPTS;
    Cmd *_data;
    Cmd *_generateData;

    virtual void initialize(); // Called by Application class to initialize
                               // the main window and to create and initialize
                               // the menuBar.

  public:

    MenuWindow ( char *name ); // Constructor.

    // Provides access to the commands in the class.
    Cmd *exitCmd()    { return _exit; }
    Cmd *openCmd()  { return _open; }
    Cmd *runCmd() { return _run; }
    Cmd *detGraphCmd() { return _detGraph; }
    Cmd *maxPllGraphCmd() { return _maxPllGraph; }
    Cmd *adjacencyMatrixCmd() { return _adjacencyMatrix; }
    Cmd *resultsCmd() {return _results;}
    Cmd *dataCmd() {return _data;}
    Cmd *generateData() {return _generateData;}

    void createMenuPanes(); // Function to create the menu panes.
    void reset();
```

```
    virtual ~MenuWindow();
};
#endif

////////////////////////////////////////////////////////////////////////////////
// MenuWindow.C: This file contains the member functions of the MainWindow class.
////////////////////////////////////////////////////////////////////////////////

#include "MenuWindow.h"
#include "CmdList.h"
#include "MenuBar.h"
#include "RunCmd.h"
#include "DetGraphCmd.h"
#include "MaxPllGraphCmd.h"
#include "AdjacencyMatrixCmd.h"
#include "ExitCmd.h"
#include "OpenCmd.h"
#include "ResultsCmd.h"
#include "HowToRunCmd.h"
#include "AboutUMPTSCmd.h"
#include "DataCmd.h"
#include "GenerateYourDataCmd.h"
#include <assert.h>

MenuWindow::MenuWindow ( char *name )   : MainWindow ( name )
{
  _menuBar = NULL;

 // Create the application-wide commands that appear in all menus
 // TRUE indicates that the Widget is initially active and FALSE indicates
 // that the Widget is initially inactive.
 _run      = new RunCmd ("Run", FALSE);
 _exit     = new ExitCmd ( "Exit" , TRUE );
 _open   = new OpenCmd ( "Open", TRUE );
 _detGraph = new DetGraphCmd("Determinate Graph", FALSE);
 _maxPllGraph = new MaxPllGraphCmd("Maximally Parallel Graph", FALSE);
 _adjacencyMatrix = new AdjacencyMatrixCmd("Adjacency Matrix", FALSE);
 _results = new ResultsCmd("Results", FALSE);
 _howToRun = new HowToRunCmd("How To Run", TRUE);
 _aboutUMPTS = new AboutUMPTSCmd("About UMPTS",TRUE);
 _data = new DataCmd("About Data", TRUE);
 _generateData = new GenerateYourDataCmd("Generating Your Data files", TRUE);

 // Set the list of command to be activated and deactivated, when the Open command is
 // executed.
 _open->addToActivationList(_run);
 _open->addToDeactivationList(_detGraph);
 _open->addToDeactivationList(_maxPllGraph);
 _open->addToDeactivationList(_adjacencyMatrix);
 _open->addToDeactivationList(_results);

 // Command to be deactivated when an error occurs during the execution of Open command.
 _open->addToResetList(_run);

 // Set the list of commands whose states need to be reverted to the previous state, if
 // execution of Open command is canceled.
 _open->addToRevertList(_run);
 _open->addToRevertList(_detGraph);
 _open->addToRevertList(_maxPllGraph);
 _open->addToRevertList(_adjacencyMatrix);
 _open->addToRevertList(_results);

 // Set the activation and deactivation list for the Run command.
 _run->addToActivationList(_detGraph);
 _run->addToActivationList(_maxPllGraph);
 _run->addToActivationList(_adjacencyMatrix);
 _run->addToActivationList(_results);

 // Set the revertList for the Run command.
 _run->addToRevertList(_detGraph);
 _run->addToRevertList(_maxPllGraph);
 _run->addToRevertList(_adjacencyMatrix);
 _run->addToRevertList(_results);

 _detGraph->addToActivationList(_run);
 _maxPllGraph->addToActivationList(_run);
```

```
 _adjacencyMatrix->addToActivationList(_run);

 _results->addToActivationList(_run);
}

// This function initializes the window, menuBar and then creates the menu panes.
void MenuWindow::initialize()
{

    // Call base class to create XmMainWindow widget
    // and set up the work area.
    MainWindow::initialize();

    _menuBar = new MenuBar(_main, "menuBar");

    // Set the initial width and height of the main window widget.
    XtVaSetValues ( _main,
                    XmNmenuBar, _menuBar->baseWidget(),
                    XmNwidth,500,
                    XmNheight,400,
                    NULL);

    // Called to add panes to the menu.
    createMenuPanes();

    // Make the menubar visible by managing it.
    _menuBar->manage();
}

// This function creates the menu panes by adding the commands to the
// menu bar. All commands are represented by pushButton widgets.
void MenuWindow::createMenuPanes()
{
 CmdList *cmdList;

 // Create an Application pane containing Open, Run,
 // and other application-wide commands

 cmdList = new CmdList();
 assert(cmdList);
 cmdList->add ( _open );           // Add the commands in the File menu to the list.
 cmdList->add ( _run );            // Each pulldown menu forms a list.
 cmdList->add ( _exit );
 _menuBar->addCommands ( cmdList, "File" );

 delete cmdList;
 cmdList = NULL;

 cmdList = new CmdList();
 cmdList->add( _adjacencyMatrix);  // Add the commands in the view menu to the list.
 cmdList->add( _detGraph);
 cmdList->add( _maxPllGraph);
 cmdList->add( _results);

 _menuBar->addCommands ( cmdList, "View" );

 delete cmdList;

 assert(cmdList = new CmdList());
 cmdList->add(_howToRun);          // Add the commands in the Help menu to the list.
 cmdList->add(_aboutUMPTS);
 cmdList->add(_data);
 cmdList->add(_generateData);

 _menuBar->addCommands( cmdList, "Help");

 cmdList = NULL;
}

// Destructor. Free memory by deleting all the commands.
MenuWindow::~MenuWindow()
{
 delete _run;
 delete _exit;
 delete _open;
```

```
 delete _detGraph;
 delete _maxPllGraph;
 delete _results;
 delete _adjacencyMatrix;
 delete _menuBar;
 delete _howToRun;
 delete _aboutUMPTS;
 delete _data;
 delete _generateData;
}

////////////////////////////////////////////////////////////////////////////////
// DialogManager.h: This class provides three types of dialogs. They are the error,
//  information and Question dialogs.
////////////////////////////////////////////////////////////////////////////////
#ifndef DIALOGMANAGER_H
#define DIALOGMANAGER_H

#include "BaseWidget.h"
enum DialogEnum{INFO,ERROR,QUESTION};

typedef void (*DialogCallback)(void *);

class DialogManager : public BaseWidget{

  private:
    DialogEnum _dialogType;
    DialogCallback _callback;
    void* _clientData;

    Widget getDialog(); // Returns a dialog box.

    // Callback functions for the dialog widget.
    static void okCallback ( Widget, XtPointer, XtPointer);

    // Called to get a new dialog.
    Widget createDialog ( Widget );

 public:

    DialogManager ( char * ,DialogEnum);
    void okSelected();

    static void destroyTempCallback(Widget, XtPointer, XtPointer);
    void post ( char *, void*, DialogCallback OK = NULL);
    ~DialogManager();
);

// The DialogManager class exports global pointers for providing
// information, error and Question dialog boxes. These dialogs are used throughout
// the program to display errors, information and to ask questions to the user.
extern DialogManager *theInfoDialogManager;
extern DialogManager *theErrorDialogManager;
extern DialogManager *theQuestionDialogManager;

#endif

////////////////////////////////////////////////////////////
// DialogManager.C: Supports dialog Widgets
////////////////////////////////////////////////////////////
#include <iostream.h>
#include "DialogManager.h"
#include "Application.h"
#include <Xm/MessageB.h>
#include <assert.h>

// Create global instances of information, error and Question dialog
// boxes, so that they can be used whenever a dialog needs to be displayed.

DialogManager *theInfoDialogManager = new DialogManager("Information", INFO);
DialogManager *theErrorDialogManager = new DialogManager("Error", ERROR);
DialogManager *theQuestionDialogManager =new DialogManager("Confirm", QUESTION);

// Constructor for the DialogManager class.
DialogManager::DialogManager(char *name, DialogEnum DialogType):BaseWidget(name)
{
```

```
    _dialogType = DialogType;
}

// Destructor.
DialogManager::~DialogManager()
{
 XtDestroyWidget(_baseWidget);
}

// This function creates a new dialog if one doesn't exist and returns it.
// If a dialog already exists, then that dialog is returned.
Widget DialogManager::getDialog()
{
    Widget newDialog = NULL;

    // If the permanent widget exists and is not in use,
    // just return it

    if ( _baseWidget && !XtIsManaged ( _baseWidget) )
        return _baseWidget;

    // Get a widget from the derived class

  newDialog = createDialog ( theApplication->baseWidget() ) ;

  if(_baseWidget)
  {
        XtAddCallback(newDialog,
                      XmNokCallback,
                      &DialogManager::destroyTempCallback,
                     (XtPointer) this);

        XtAddCallback(newDialog,
                      XmNcancelCallback,
                      &DialogManager::destroyTempCallback,
                      (XtPointer) this);
  }
  else
   _baseWidget = newDialog;

 return newDialog;
}

// Creates a new dialog based on the dialog type.
Widget DialogManager::createDialog(Widget BaseWidget)
{
   Widget dialog;
   XmString Ok, Cancel;

   XmString Title = XmStringCreateLtoR(_baseName, XmSTRING_DEFAULT_CHARSET);

   switch(_dialogType)
   {
         case INFO:
                // Create an Information dialog.
                dialog = XmCreateInformationDialog(BaseWidget,_baseName,NULL,0);

                Ok = XmStringCreateLtoR("OK", XmSTRING_DEFAULT_CHARSET);

                XtVaSetValues(dialog,
                XmNokLabelString, Ok,
                XmNdialogTitle, Title,
                XmNautoUnmanage, TRUE,
                NULL);

                // Unmanage the cancel button. Since the Information dialog does
                // not need one.
                XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_CANCEL_BUTTON));

                break;

         case ERROR:
                // Create an Error dialog.
                dialog = XmCreateErrorDialog(BaseWidget,_baseName,NULL,0);

                // Create the string OK.
```

```
                Ok = XmStringCreateLtoR("OK", XmSTRING_DEFAULT_CHARSET);

                // Set the resources for the dialog.
                XtVaSetValues(dialog,
                              XmNokLabelString, Ok,
                              XmNdialogTitle, Title,
                              XmNautoUnmanage, TRUE,
                              NULL);
                // The Error dialog box does not have a cancel button.
                // So unmanage the cancel button.
                XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_CANCEL_BUTTON));
                break;

        case QUESTION:
                // Create a Question dialog.
                dialog = XmCreateQuestionDialog(BaseWidget,_baseName,NULL,0);

                // Create Ok and Cancel Strings to be displayed in the dialog.
                Ok = XmStringCreateLtoR("Yes", XmSTRING_DEFAULT_CHARSET);
                Cancel = XmStringCreateLtoR("No", XmSTRING_DEFAULT_CHARSET);

                // Set the resource values for the dialog.
                XtVaSetValues(dialog,
                              XmNokLabelString, Ok,
                              XmNcancelLabelString, Cancel,
                              XmNdialogTitle, Title,
                              XmNautoUnmanage, TRUE,
                              NULL);

                break;
    }

    // Free the memory allocated for the strings.
    XmStringFree(Ok);
    XmStringFree(Title);

    // Unmanage the "Help" button since it is not used.
    XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON));

    return dialog;
}

// This function posts the string passed to it into the dialog box.
void DialogManager::post ( char *text1, void* clientData, DialogCallback ok)
{
    // Get a dialog widget

    Widget dialog = getDialog();

    _callback = ok;
    _clientData = clientData;

    // Make sure the dialog exists, and that it is an XmMessageBox
    // or subclass, since the callbacks assume this widget type
    assert ( dialog );
    assert ( XtIsSubclass ( dialog, xmMessageBoxWidgetClass ) );

    // Set the dialog style to FULL_APPLICATION_MODAL, so that nothing else
    // in the application can be done before responding to the dialog
    XtVaSetValues(dialog,
                  XmNdialogStyle, XmDIALOG_FULL_APPLICATION_MODAL,
                  NULL);

    // Convert the text string to a compound string and
    // specify this to be the message displayed in the dialog.
     XmString str = XmStringCreateLtoR( text1 ,XmSTRING_DEFAULT_CHARSET);

     XtVaSetValues ( dialog, XmNmessageString, str, NULL );

    // Free the memory allocated for str.
    XmStringFree ( str );

    XtAddCallback( dialog,
                  XmNokCallback, &DialogManager::okCallback,
                  (XtPointer) this);
```

```
    XtManageChild(dialog);
}

// Destroy the temporary dialog boxes created.
void DialogManager::destroyTempCallback ( Widget w, XtPointer, XtPointer)
{
    XtDestroyWidget ( w );
}

// Callback function executed when the OK Button is selected.
void DialogManager::okCallback ( Widget    W, XtPointer clientData, XtPointer )
{
    DialogManager      *obj = (DialogManager *) clientData;

    if(obj->_dialogType == QUESTION)
    obj->okSelected();

    if(obj->baseWidget() != W)
    XtDestroyWidget(W);

}

void DialogManager::okSelected()
{
 if(_callback)
 _callback(_clientData);
}

#ifndef FSDIALOG_H
#define FSDIALOG_H
#include "BaseWidget.h"

////////////////////////////////////////////////////////////////////////
// FileSelectionDialog.h
// The FileCelectionDialog class implements the FileSelectionDialog
// class. When the user selects the Open command, the FileSelectionDialog
// box is displayed.
////////////////////////////////////////////////////////////////////////
typedef void(*FileSelectionCallback) (void*, char*);
class FileSelectionDialog: public BaseWidget
{
 private:

    // callback functions
    static void okCallback(Widget, XtPointer, XtPointer);
    static void cancelCallback(Widget, XtPointer, XtPointer);
    static void helpCallback(Widget, XtPointer, XtPointer);
public:
    FileSelectionDialog(char*, FileSelectionCallback, void*);
    FileSelectionCallback _callback; // Function to be called after the file
                                     // is selected.

    virtual ~FileSelectionDialog(void);
    void* _clientData;
    void fileSelected(char*);
    virtual void manage(void);       // Manages the widget.
};
#endif

#include "FileSelectionDialog.h"
#include <sys/types.h>
#include <string.h>
#include <dirent.h>
#include <iostream.h>
#include <X11/Xos.h>
#include "Application.h"
#include "DialogManager.h"
#include <Xm/FileSB.h>

////////////////////////////////////////////////////////////////////////
//The constructor first creates a file selection box and then
// registers callbacks. callback is the callback function to be
// called after the user selects a file. This function is in OpenCmd
// class.
////////////////////////////////////////////////////////////////////////
FileSelectionDialog::FileSelectionDialog(char* name,
```

```
                        FileSelectionCallback callback, void* clientData):BaseWidget(name)
{
    _clientData = clientData;
    _callback = callback;

    // Create a file selection dialog box.
    _baseWidget = XmCreateFileSelectionDialog(theApplication->baseWidget(),name, NULL, 0);

    // Call handleDestruction function in BaseWindow, to take care of freeing memory
    // and avoid dangling pointers when the Widget is destroyed.
    handleDestruction();

    // Sets the initial directory to read from. Only filenames with the dat extension
    // are displayed. Sets the dialog style to FULL_APPLICATION_MODAL.
    XtVaSetValues(_baseWidget,
              XmNdirectory, XmStringCreateSimple("/g/akamala/thesis/GUI/"),
              XmNdirMask, XmStringCreateSimple("/g/akamala/thesis/GUI/*.dat"),
              XmNdialogStyle, XmDIALOG_FULL_APPLICATION_MODAL,
              NULL);

   // Set callback functions to be called when the Ok, Cancel and
   // Help buttons are pressed.
   XtAddCallback(_baseWidget,
                 XmNcancelCallback, &FileSelectionDialog::cancelCallback,
                 (XtPointer)this);

   XtAddCallback(_baseWidget,
                 XmNokCallback, &FileSelectionDialog::okCallback,
                 (XtPointer)this);

   XtAddCallback(_baseWidget,
                  XmNhelpCallback, &FileSelectionDialog::helpCallback,
                  (XtPointer)this);

   XtManageChild(_baseWidget);
}


//Destructor.
FileSelectionDialog::~FileSelectionDialog()
{
 XtDestroyWidget(_baseWidget);
}

// Callback function to be called when the Ok button is pressed.
void FileSelectionDialog::okCallback(Widget, XtPointer clientData,
                                XtPointer callData)
{
   FileSelectionDialog* obj = (FileSelectionDialog*) clientData;

   XmFileSelectionBoxCallbackStruct* cb =
   (XmFileSelectionBoxCallbackStruct*)callData;
   char* fileName = NULL;
   XmString xmstr = cb->value;
   int status=0;
   int i=0;

   if(xmstr)
   {
     // Convert XmString into a character string format.
     status = XmStringGetLtoR(xmstr, XmSTRING_DEFAULT_CHARSET, &fileName);
     if(!status)() // If no file is selected.
     else
      obj->fileSelected(fileName);
   }
   // Make the FileSelectionDialog box invisible by unmanaging it.
   obj->BaseWidget::unmanage();
}

// Cancel button is pressed.
void FileSelectionDialog::cancelCallback(Widget, XtPointer clientdata, XtPointer)
{
   FileSelectionDialog* obj = (FileSelectionDialog*) clientdata;
   obj->fileSelected((char*)NULL);
   obj->BaseWidget::unmanage();
}
```

```
// Provide information on which file to select.
void FileSelectionDialog::helpCallback(Widget, XtPointer, XtPointer)
{
   char str1[100], str2[300];

   strcpy(str2," -Select a Random data file (Ex: Random1.dat) or \n");
   strcpy(str1,"  a Fixed data file (Ex:Fixed1.dat).\n\n");
   strcat(str2,str1);
   strcpy(str1," -For the task systems in the data files, random\n");
   strcat(str2,str1);
   strcpy(str1,"  domains and ranges are generated and hence the results\n");
   strcat(str2,str1);
   strcpy(str1,"  obtained may not be the same for each run.\n\n");
   strcat(str2,str1);
   strcpy(str1," -For the example files, fixed domains and ranges are used\n");
   strcat(str2,str1);
   strcpy(str1,"  and hence the results obtained for each run are the same.\n");
   strcat(str2,str1);

   // Display the information.
   theInfoDialogManager->post(str2, (void*)NULL);

}

// Passes the fileName to the appropriate function.
void FileSelectionDialog::fileSelected(char* fileName)
{
  if(_callback)            // Pass the filename to the appropriate function.
  _callback(_clientData, fileName);
}

// Manage the file Selection dialog box.
void FileSelectionDialog::manage()
{
   BaseWidget::manage();
   XtPopup(XtParent(_baseWidget),XtGrabNone);
   XMapRaised(theApplication->display(),XtWindow(_baseWidget));
}

//////////////////////////////////////////////////////////////////////////
// Cmd.h: The Cmd class acts as the  base class for all command objects.
//  It maintains an activation list and a deactivation list.
//  Activation list is the list of commands to be activated when the
//  command is executed and the deactivation list is the list of commands
//  to be deactivated when the command is executed.
//////////////////////////////////////////////////////////////////////////
#ifndef CMD_H
#define CMD_H

class CmdList;
class CmdButtonInterface;

class Cmd {

    friend CmdButtonInterface;

  private:

    // Lists of other commands to be activated or deactivated
    // when this command is executed or "undone".
    CmdList       *_activationList;
    CmdList       *_deactivationList;

    // The resetList indicates the commands to be deactivated when the
    // command is selected and then canceled.
    // For Ex., if the Open command is selected and if an invalid file is
    // selected, then the commands in the reset list are deactivated.
    CmdList       *_resetList;

    // The revertList contains the list of commands to be sent back to
    // the previous state.
    // For Ex., if the cancel button is selected in the file selection
    // dialog box, then all the commands activated and deactivated due
    // to the activation of the Open command needs to be reverted back
    // to the previous state. The revertList contains these commands.
```

```
    CmdList       *_revertList;

    int            _active;           // Is this command currently active?.
    int            _previouslyActive; // Previous value of _active.
    char          *_name;             // Name of this Cmd.
    CmdButtonInterface **_ci;
    int            _numInterfaces;    // Number of interfaces for commands.

  protected:

    // This is a pure virtual function and derived classes need to implement it for
    virtual void doit()   = 0;     // performing actions represented by that class.

    virtual void revert();         // Reverts object to previous state.
    void revertList();

  public:

    Cmd ( char *,  int );          // Protected constructor.

    virtual ~Cmd ();               // Destructor.

    // Public interface for executing and undoing commands.

    virtual void execute();

    void    activate();            // Activate this object.
    void    deactivate();          // Deactivate this object.
    void    reset();               // Reset the associated commands.

    // Functions to register dependent commands.
    void    addToActivationList ( Cmd * );
    void    addToDeactivationList ( Cmd * );
    void    addToResetList( Cmd *);
    void    addToRevertList( Cmd *);

    // Register a Widget used to execute this command.
    void    registerInterface ( CmdButtonInterface * );

    // Access functions.

    int active () { return _active; }     // Returns the current state of the command.
    const char *const name () { return _name; }
    virtual const char *const className () { return "Cmd"; }
};
#endif

//////////////////////////////////////////////////////////////////////
// Cmd.C: This file contains the member functions of the Cmd class.
//////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include "Cmd.h"
#include "CmdList.h"
#include "CmdButtonInterface.h"

Cmd::Cmd ( char *name, int active )
{
    // Initialize all data members.

    _name              = name;
    _active            = active; // _active can be 1 or 0. When it is 1 the command is
                                 // active, i.e., it can be executed. When _active is 0
                                 // the command is inactive, i.e., it appears grayed and
                                 // cannot be executed.
    _numInterfaces     = 0;
    _ci                = NULL;
    _activationList    = NULL;
    _deactivationList  = NULL;
}

// Destructor.
Cmd::~Cmd()
{
    delete _activationList;
    delete _deactivationList;
    delete _ci;
```

```
    _activationList = NULL;
    _deactivationList = NULL;
    _ci = NULL;
}

// Register the pushButton interface to the command.
void Cmd::registerInterface ( CmdButtonInterface *ci )
{
    // Make a new list, large enough for the new object.

    CmdButtonInterface **newList = new CmdButtonInterface*[_numInterfaces + 1];

    // Copy the contents of the previous list to
    // the new list.

    for( int i = 0; i < _numInterfaces; i++)
          newList[i] = _ci[i];

    // Free the old list.
    delete []_ci;

    // Install the new list.
    _ci =  newList;

    // Add the object to the list and update the list size.

    _ci[_numInterfaces] = ci;
    _numInterfaces++;

    if ( ci )
      if ( _active )      // If the command is active, activate the push button
        ci->activate();  // represeinting else deactivate the push button.
      else
        ci->deactivate();
}

// Activate the command.
void Cmd::activate()
{
    // Activate the associated interfaces.
    for ( int i = 0; i < _numInterfaces; i++ )
       _ci[i]->activate ();

    // Save the current value of _active before setting the new state.
    // _active is a flag, which when set indicates that the command is active.
    // In this way _previouslyActive which contains the previous state of the command
    // can be used to revert the command to its previous state.
    _previouslyActive = _active;
    _active = TRUE;
}

// Deactivate the command. A command that is not active appears
// gray in color.
void Cmd::deactivate()
{
    // Deactivate the associated interfaces
        for ( int i = 0; i < _numInterfaces; i++ )
       _ci[i]->deactivate ();

    // Save the current value of active before setting the new state, so that
    // _previouslyActive can be used to revert the command to its previous state
    // when the operation associated with command is canceled.
    _previouslyActive = _active;
    _active = FALSE;
}

// Reset the commands in the command's _resetList.
void Cmd::reset()
{
   if ( _resetList )
        for ( int i = 0; i < _resetList->size(); i++ )
        (*_resetList)[i]->deactivate();
}

// This function reverts the commands in the list to the previous state.
```

```
void Cmd::revertList()
{
 // If there are commands to be reverted, revert them to their previous state.
 if ( _revertList )
 {
   for ( int i = 0; i < _revertList->size(); i++ )
   (*_revertList)[i]->revert();
  }
}

//reset the command to the previous state
void Cmd::revert()
{
    // Activate or deactivate, as necessary,
    // to return to the previous state.

    if ( _previouslyActive )
    activate();
    else
    deactivate();
}

//  Add a command to the revertList of the command.
void Cmd::addToRevertList(Cmd *cmd)
{
  if(!_revertList ) // If memory is not allocated, allocate memory.
_revertList = new CmdList();

_revertList->add(cmd);
}

// Add command passed as argument to the list to be activated, when the
// current command is executed.
void Cmd::addToActivationList ( Cmd *cmd )
{
    if ( !_activationList )
       _activationList = new CmdList();

    _activationList->add ( cmd );
}

// Add command passed as argument to the list to be deactivated, when the
// current command is executed.
void Cmd::addToDeactivationList ( Cmd *cmd )
{
    if ( !_deactivationList )
     _deactivationList = new CmdList();

    _deactivationList->add ( cmd );
}

// Add command passed as argument to the list to be deactivated, when an
// error occurs during the execution of the command.
void Cmd::addToResetList( Cmd *cmd)
{
 if(!_resetList )
 _resetList = new CmdList();

 _resetList->add( cmd );
}

// This function is executed when the command represented by a derived
// class is executed. This calls the derived class's doit() member function
// to perform the desired action.
void Cmd::execute()
{
    int i;

    // If a command is inactive, it cannot be executed.
    if ( !_active )
       return;


    // Activate all the commands in the _activationList.
    if ( _activationList )
     for ( i = 0; i < _activationList->size(); i++ )
```

```
     (*_activationList)[i]->activate();

    // Deactivate all the commands in the _deactivationList.
    if ( _deactivationList )
     for ( i = 0; i < _deactivationList->size(); i++ )
     (*_deactivationList)[i]->deactivate();

    //Call the derived class's doit() member function to
    // perform the action represented by this object.
     doit();
}

//////////////////////////////////////////////////////
// CmdButtonInterface.h
// The CmdButtonInterface.h gives a push button interface
// to the Cmd Class.
//////////////////////////////////////////////////////
#ifndef CMDINTERFACE_H
#define CMDINTERFACE_H
#include "BaseWidget.h"

class Cmd;

class CmdButtonInterface : public BaseWidget{

    friend Cmd;

  private:

    Cmd    *_cmd;
    int _active; // Flag indicating whether the command is active or not.

  public:

     // callback function to be called when the button representing the command
     // is activated.
     static void buttonPressCallback ( Widget, XtPointer, XtPointer);


    // Constructor.
    CmdButtonInterface ( Widget, Cmd * );

    virtual void activate(); // Activate the pushButton widget representing
                             // the command.
    virtual void deactivate(); //Gray the pushButton widget.
};
#endif

//////////////////////////////////////////////////////
// CmdButtonInterface.C
//////////////////////////////////////////////////////
#include "CmdButtonInterface.h"
#include "Cmd.h"
#include <Xm/PushB.h>

CmdButtonInterface::CmdButtonInterface ( Widget parent, Cmd *cmd ) : BaseWidget( cmd-
>name() )
{
     XFontStruct *font = NULL;
     XmFontList fontList = NULL;
     char* nameString = NULL;
     XmStringCharSet char_set = XmSTRING_DEFAULT_CHARSET;

     // Create the pushButton.
     _baseWidget = XtCreateWidget( _baseName,
                                   xmPushButtonWidgetClass,
                                   parent,
                                   NULL,0);

    // Load the needed font
    nameString = "-ncd*medium*11*";
    font = XLoadQueryFont(XtDisplay(_baseWidget),nameString);
    fontList = XmFontListCreate(font, char_set);

    XtVaSetValues(_baseWidget,
                  XmNfontList, fontList,
```

```
                    NULL);

    _active = TRUE;
    _cmd    = cmd;

    // If the _active flag is TRUE activate the command else deactivate it.
    if(_active)
       activate();
    else
        deactivate();

    // Register a callback function to be called when the button is pressed.
    XtAddCallback(_baseWidget,
                  XmNactivateCallback,
                  &CmdButtonInterface::buttonPressCallback,
                  (XtPointer) this);

    // Register the interface with the Cmd class.
    cmd->registerInterface ( this );
}

// Callback to be called when the Command is selected.
void CmdButtonInterface::buttonPressCallback ( Widget,
                                               XtPointer clientData,
                                               XtPointer )
{
    CmdButtonInterface *obj = (CmdButtonInterface *) clientData;

    obj->_cmd->execute();
}

// Activate the button, so that the command can be executed.
void CmdButtonInterface::activate()
{
    if ( _baseWidget )
        XtSetSensitive ( _baseWidget, TRUE );
    _active = TRUE;
}

// Gray the command, i.e., deactivate it.
void CmdButtonInterface::deactivate()
{
    if ( _baseWidget)
        XtSetSensitive ( _baseWidget, FALSE );

    // Deactivate the command by setting the _active flag to FALSE.
    _active = FALSE;
}

//////////////////////////////////////////////////////////////
// CmdList.h: Maintain a list of Cmd objects
//////////////////////////////////////////////////////////////

class Cmd;

class CmdList {

private:

    Cmd **_contents;                        // The list of objects.
    int   _numElements;                     // Current size of list.

public:

    CmdList();                              // Construct an empty list.
    virtual ~CmdList();                     // Destroys list, but not objects in list.

    void add ( Cmd * );                     // Add a single Cmd object to list.

    Cmd **contents() { return _contents; } // Returns the list.
    int size() { return _numElements; }     // Returns list size.
    Cmd *operator[]( int );                 // Returns an element of the list.

};
```

```
//////////////////////////////////////////////////////////////
// CmdList.C: The CmdList class maintains a list of Cmd objects.
//////////////////////////////////////////////////////////////
#include "CmdList.h"

class Cmd; // Forward reference.

// Constructor.
CmdList::CmdList()
{
    // The list is initially empty

    _contents    = 0;
    _numElements = 0;
}

// Destructor.
CmdList::~CmdList()
{
    // free the list
    delete []_contents;
}

//////////////////////////////////////////////////////////////
//This function adds commands to the list
//////////////////////////////////////////////////////////////
void CmdList::add ( Cmd *cmd )
{
    int i;
    Cmd **newList;

    // Allocate a list large enough for one more element

    newList = new Cmd*[_numElements + 1];

    // Copy the contents of the previous list to
    // the new list

    for( i = 0; i < _numElements; i++)
        newList[i] = _contents[i];

    // Free the old list

    delete []_contents;

    // Make the new list the current list

    _contents =  newList;

    // Add the command to the list and update the list size.

    _contents[_numElements] = cmd;

    _numElements++;
}

Cmd *CmdList::operator[] ( int index )
{
    // Return the indexed element
    return _contents[index];
}

//////////////////////////////////////////////////////////////
// OpenCmd.h: OpenCmd class is used to open an input file.
//  It has a pointer to the FileSelectionDialog class as a
//  private data member. When the Open command is activated,
//  a fileselection dialog  box is displayed.
//////////////////////////////////////////////////////////////

#ifndef OPENCMD_H
#define OPENCMD_h
#include "Cmd.h"
#include "UMPTSApp.h"
#include "FileSelectionDialog.h"
#include "DialogManager.h"
```

```
class OpenCmd : public Cmd{
 private:
    // Pointer to a FileSelectionDialog class.
    FileSelectionDialog* _fsDialog;

     static void readFileCallback(void*, char*);
 protected:
     // Called from the Cmd class's execute member function.
     virtual void doit();
     public:
     OpenCmd(char*, int); // Constructor,
     virtual void execute();
     virtual const char *const className(){ return "OpenCmd";}
     void readFile(char*);
     ~OpenCmd(void);
};
#endif

#include "OpenCmd.h"
#include "MenuWindow.h"
#define TRUE 1

// Constructor
OpenCmd::OpenCmd(char* name, int active): Cmd(name, active)
{
    _fsDialog = NULL;
}

// This function is called when the Open Command is activated.
void OpenCmd::execute()
{
    // Call the execute function in the Cmd class.
    Cmd::execute();
}

// This function is called from the execute() member function
// in the Cmd class. This overrides the doit() pure virtual
// function in Cmd class.
void OpenCmd::doit()
{

    // The function readFileCallback is passed to the constructor
    // of the FileSelectionDialog class.
    _fsDialog = new FileSelectionDialog("File Selection",
    &OpenCmd::readFileCallback,(void*)this);
    _fsDialog->manage();
}

// This function is called from the FileSelectionDialog class.
// The argument FileName indicates the name of the file selected
// in the fileSelectionDialog box.
void  OpenCmd::readFileCallback(void* clientData, char* FileName)
{
    OpenCmd* obj = (OpenCmd*) clientData;
    obj->readFile(FileName);
}

// This function checks the validity of the file selected.
// If a wrong file is selected, then and error is displayed.
// If a correct file is selected, then it is passed to the UMPTSClass.
void OpenCmd::readFile(char* FileName)
{
    char str[100];
    char *temp1, *temp2;

    strcpy(str, FileName);

    strtok(str,"/");
    while(temp1 = strtok(NULL,"/"))
    temp2 = temp1;

    if(FileName == NULL) // If cancel is selected in the fileSelectionBox
    revertList();        // revert all the commands to their previous state.
    else
    if( (strspn(temp2,"Random") == 6) || (strspn(temp2,"Fixed") == 5) )
    {
```

```
      theUMPTSApp->setFileName(FileName,temp2);

       strcpy(str, "Select Run in the File menu to run the algorithm\n");
       strcat(str, "or Select Open to open another input file.");

      theInfoDialogManager->post(str,(void*)this);
    }
    else
    {
      strcpy(str, "Invalid file chosen.!\nChoose a Random or a Fixed data file.\n");
      theErrorDialogManager->post(str,(void*)this);

      reset(); // Deactivate all the commands in the resetList.
    }

    delete _fsDialog;
    _fsDialog = NULL;
}

// Destructor.
OpenCmd::~OpenCmd()
{
    delete _fsDialog;
}

////////////////////////////////////////////////////////////////
// RunCmd.h : Runs the maximally parallel task system generator.
////////////////////////////////////////////////////////////////
#ifndef RUNCMD_H
#define RUNCMD_H
#include "Cmd.h"
#include <iostream.h>
#include "Application.h"

class RunCmd : public Cmd {

 protected:

    virtual void doit();      // Call the function to run the algorithm.

 public:

    RunCmd( char *, int );
    virtual void execute();   //overrides the execute in Cmd class.
    virtual const char *const className () { return "RunCmd"; }
};
#endif

////////////////////////////////////////////////////////////
// RunCmd.C: Executes the UMPTS algorithm.
////////////////////////////////////////////////////////////
#include "RunCmd.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <X11/cursorfont.h>
#include "UMPTSApp.h"
#include "MainWindow.h"
#include "DialogManager.h"

// Constructor
RunCmd::RunCmd( char *name, int active ) :
                    Cmd ( name, active )
{
 //Empty
}

// This function is called when the RunCmd is executed.
void RunCmd::execute()
{
    Cmd::execute();
}

// This function is called from the execute function in the Cmd class.
// This function runs the maximal parallelism generation process. It
// changes the cursor shape to indicate that processing is going on.
```

```
void RunCmd::doit()
{
    int i=1;
    char str1[100], str2[600];
    int Fail =0;

    static Cursor cursor;
    XSetWindowAttributes attrs;
    Display *display; // A pointer to the Display structure.

    // Obtain the display of the window.
    display = XtDisplay(theMainWindow->baseWidget());

    // Declare the cursor to show a watch.
    cursor = XCreateFontCursor(display, XC_watch);
    attrs.cursor = cursor;

    // Change the cursor to display a watch to indicate that the application is busy.
    XChangeWindowAttributes(display, XtWindow(theMainWindow->baseWidget()),
                            CWCursor, &attrs);

    XFlush (display);

    // run the algorithm in response to the activation of the command.
    if(theUMPTSApp->runAlgorithm() == Fail)
    {
         revertList();
         strcpy(str1,"Error in the input file!. Refer to the Help menu,\n");
         strcpy(str2,"to know more about creating your own fixed data files.\n");
         strcat(str1,str2);
         theErrorDialogManager->post(str1,(void*)this);

         // Now that the execution of the algorithm is over, change the cursor back to
         // its normal state.
         attrs.cursor = None;
         XChangeWindowAttributes(display, XtWindow(theMainWindow->baseWidget()),
                                 CWCursor, &attrs);

    return;
    }

    // Change the cursor back to normal.
    attrs.cursor = None;
    XChangeWindowAttributes(display, XtWindow(theMainWindow->baseWidget()),
                            CWCursor, &attrs);

    strcpy(str2,"Finished generating the Maximally Parallel Task System.\n\n");
    strcpy(str1,"In the View menu, select one of the following:\n");
    strcat(str2,str1);
    strcpy(str1," -Select Adjacency Matrix, to view the adjacency\n");
    strcat(str2,str1);
    strcpy(str1,"  matrices of the determinate and the corresponding\n");
    strcat(str2,str1);
    strcpy(str1,"  maximally parallel task systems.\n\n");
    strcat(str2,str1);

    strcpy(str1," -Select Determinate Graph, to view the precendence\n");
    strcat(str2,str1);
    strcpy(str1,"  graph of the determinate task system.\n\n");
    strcat(str2,str1);

    strcpy(str1," -Select Maximally Parallel Graph, to view the precedence\n");
    strcat(str2,str1);
    strcpy(str1,"  graph of the maximally parallel task system\n\n");
    strcat(str2,str1);

    strcpy(str1," -Select Results to view the comparisons between the\n");
    strcat(str2,str1);
    strcpy(str1,"  determinate and the corresponding maximally parallel\n");
    strcat(str2,str1);
    strcpy(str1,"  task systems.\n");
    strcat(str2,str1);

        theInfoDialogManager->post(str2,(void*)this);
}
```

```
//////////////////////////////////////////////////////////
// ExitCmd.h: Exit an application after checking with user.
//////////////////////////////////////////////////////////
#ifndef QUITCMD_H
#define QUITCMD_H
#include "Cmd.h"
#include <iostream.h>

class ExitCmd : public Cmd {

 protected:

    virtual void doit();              // Call exit.
    static void yesCallback(void *); // Function to be executed when Yes
                                      // is selected in the dialog box.
 public:

    ExitCmd ( char *, int );
    virtual void execute();//overrides the execute in Cmd class
    virtual const char *const className () { return "ExitCmd"; }
};
#endif

//////////////////////////////////////////////////////////
// ExitCmd.C: Exit an application after checking with user.
//////////////////////////////////////////////////////////
#include "ExitCmd.h"
#include <stdlib.h>
#include "DialogManager.h"

ExitCmd::ExitCmd ( char *name, int active ) :
                    Cmd ( name, active )
{
   // Empty
}


void ExitCmd::execute()
{

    // Post a message confirming the exit operation.
    theQuestionDialogManager->post("Do you really want to exit?",
                                   (void*) this,
                                   &ExitCmd::yesCallback);


}

// If yes is selected, exit.
void ExitCmd::yesCallback(void *clientData)
{
    ExitCmd *obj = (ExitCmd *) clientData;
    obj->Cmd::execute();      // This calls the Cmd class's execute() member function.
}

void ExitCmd::doit()
{
    // Just exit
    exit ( 0 );
}

//////////////////////////////////////////////////////////
// AdajacencyMatrixCmd.h : Command that displays the adjacency matrices
//////////////////////////////////////////////////////////
#ifndef ADJACENCYCMD_H
#define ADJACENCYCMD_H
#include "Cmd.h"
#include "MatrixWindow.h"
#include <iostream.h>

class AdjacencyMatrixCmd : public Cmd {

 private:
    MatrixWindow *_detMatrix; // Pointers to MatrixWindow representing
    MatrixWindow *_maxPllMatrix; // matrices of determinate and maximally
                                 // parallel task systems.
```

```
 protected:

    virtual void doit();      // Display the adjacency matrix.
 public:

    AdjacencyMatrixCmd( char *, int );
    virtual void execute();  //overrides the execute in Cmd class
    virtual const char *const className () { return "AdjacencyMatrixCmd"; }
    ~AdjacencyMatrixCmd();
};
#endif

////////////////////////////////////////////////////////////////////
// AdjacencyMatrixCmd.C: This file contains the member functions
// of the AdjacencyMatrixCmd class.
////////////////////////////////////////////////////////////////////
#include "AdjacencyMatrixCmd.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

// Constructor
AdjacencyMatrixCmd::AdjacencyMatrixCmd( char *name, int active ) :
                                Cmd ( name, active )
{
    if(_detMatrix)             // If a previously created _detMatrix exists,
        delete  _detMatrix;    // delete it and create a new instance.
    _detMatrix = new MatrixWindow("Determinate Task System");;

    if(_maxPllMatrix)          // If a previously created _maxPllMatrix exists,
       delete _maxPllMatrix; // delete it and create a new instance.
    _maxPllMatrix = new MatrixWindow("Maximally Parallel Task System");
}

void AdjacencyMatrixCmd::execute()
{
 Cmd::execute();
}

////////////////////////////////////////////////////////////
// This function is called from the Cmd::execute() member
// function. Initialize the window and display the matrix.
////////////////////////////////////////////////////////////
void AdjacencyMatrixCmd::doit()
{
 int OUTOFBOUND = 1;

    // If the matrix size exceeds the size of the screen, display
    // return.
    if(_detMatrix->initialize() == OUTOFBOUND)
    return;
    _detMatrix->showMatrix(DETERMINATE); // Show the matrix.
    _detMatrix->manage();                // Manage the window and it's children.

    // If the determinate matrix fits in the screen, then the
    // matrix of the maximally parallel task system will fit as well.
    _maxPllMatrix->initialize();         // Initialize the window.
    _maxPllMatrix->showMatrix(MAXPLL);   // Show the matrix.
    _maxPllMatrix->manage();             // Manage the window and it's children.

}

// Destructor.
AdjacencyMatrixCmd::~AdjacencyMatrixCmd()
{
    delete _detMatrix;
    delete _maxPllMatrix;
}

#ifndef HEAD_H
#define HEAD_H
enum TaskSysEnum { DETERMINATE = 0, MAXPLL = 1};

#endif;
```

```
//////////////////////////////////////////////////////////////
// DetGraphCmd.h : This command when activated displays the
// precedence graph of the determinate task system.
//////////////////////////////////////////////////////////////
#ifndef DETGRAPHCMD_H
#define DETGRAPHCMD_H
#include "Cmd.h"
#include <iostream.h>
#include "UMPTSApp.h"

class DetGraphCmd : public Cmd {

  protected:

    virtual void doit();      // Display the graph.
       public:

    DetGraphCmd( char *, int );
    virtual void execute();//overrides the execute in Cmd class
    virtual const char *const className () { return "DetGraphCmd"; }
};
#endif

//////////////////////////////////////////////////////////////////////////////
// DetGraphCmd.C: The DetGraphCmd class displays the precedence graph of the
//  determinate task system.
//////////////////////////////////////////////////////////////////////////////
#include "DetGraphCmd.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "head.h"

// Constructor.
DetGraphCmd::DetGraphCmd( char *name, int active ) :
                    Cmd ( name, active )
{
    //Empty
}

void DetGraphCmd::execute()
{
    Cmd::execute();
}

// Display the graph by calling the draw function in the UMPTSApp class.
void DetGraphCmd::doit()
{
    theUMPTSApp->draw(DETERMINATE);
}

//////////////////////////////////////////////////////////////
// MaxPllGraphCmd.h : This command when activated displays the
// precedence graph of the maximally parallel task system.
//////////////////////////////////////////////////////////////
#ifndef MAXPLLGRAPHCMD_H
#define MAXPLLGRAPHCMD_H
#include "Cmd.h"
#include <iostream.h>
#include "UMPTSApp.h"

class MaxPllGraphCmd : public Cmd {

  protected:

    virtual void doit();      // Display the graph.

  public:

    MaxPllGraphCmd( char *, int );

    virtual void execute();  // This function overrides the execute() in the Cmd class.
    virtual const char *const className () { return "MaxPllGraphCmd"; }
};
#endif
```

```
//////////////////////////////////////////////////////////////////////////
// MaxPllGraphCmd.C: This command when activated, displays the precedence
//  graph of the determinate task system.
//////////////////////////////////////////////////////////////////////////
#include "MaxPllGraphCmd.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "head.h"


MaxPllGraphCmd::MaxPllGraphCmd( char *name, int active ) :
                    Cmd ( name, active )
{
    //Empty
}

void MaxPllGraphCmd::execute()
{
 Cmd::execute();
}

// This function displays the graph of the maximally parallel task system.
// This function overrides the pure virtual function doit() member function
// in the Cmd class.
void MaxPllGraphCmd::doit()
{
  theUMPTSApp->draw(MAXPLL);
}

//////////////////////////////////////////////////////////////
// ResultsCmd.h : Displays the results.
//////////////////////////////////////////////////////////////
#ifndef RESULTSCMD_H
#define RESULTSCMD_H
#include "Cmd.h"
#include "ResultsWindow.h"
#include <iostream.h>

class ResultsCmd : public Cmd {
  private:
    ResultsWindow *_result;    // Pointer to the ResultsWindow class.

 protected:
    virtual void doit();       // Display the results.
 public:

    ResultsCmd( char *, int );
    virtual void execute();    // Overrides the execute in Cmd class
    virtual const char *const className () { return "ResultsCmd"; }
    ~ResultsCmd();
};
#endif

//////////////////////////////////////////////////////////
// ResultsCmd.C: Displays the comparison between the
// determinate and the maximally parallel task systems.
//////////////////////////////////////////////////////////
#include "ResultsCmd.h"

// Constructor
ResultsCmd::ResultsCmd( char *name, int active ) :
                    Cmd ( name, active )
{
    if(_result)
    delete _result;

    _result = new ResultsWindow; // Allocate memory for the pointer.
}

void ResultsCmd::execute()
{
    Cmd::execute();
}

// Overrides the pure virtual function doit() in Cmd class.
```

```
// This function initializes the window and displays the comparisons
// of the determinate and the maximally parallel task systems.`
void ResultsCmd::doit()
{
    _result->initialize(); // Initialize the window.
    _result->setResults(); // Display the results in the window.
    _result->manage();     // Manage the Window showing the results.

}

// Destructor.
ResultsCmd::~ResultsCmd()
{
    delete _result;
}

////////////////////////////////////////////////////////////
// AboutUMPTSCmd.h: Exit an application after checking with user.
////////////////////////////////////////////////////////////
#ifndef UMPTSCMD_H
#define UMPTSCMD_H
#include "Cmd.h"
#include <iostream.h>

class AboutUMPTSCmd : public Cmd {

public:

    AboutUMPTSCmd ( char *, int );
    void doit();
    virtual void execute();   // Overrides the execute in Cmd class.
    virtual const char *const className () { return "AboutUMPTSCmd"; }
};
#endif

////////////////////////////////////////////////////////////
// AboutUMPTSCmd.C: Exit an application after checking with user.
////////////////////////////////////////////////////////////
#include "AboutUMPTSCmd.h"
#include <stdlib.h>
#include "DialogManager.h"
#define TRUE 1

AboutUMPTSCmd::AboutUMPTSCmd ( char *name, int active ) :
                    Cmd ( name, active )
{
// Empty.
}

void AboutUMPTSCmd::execute()
{
    Cmd::execute(); // Call the execute() function in Cmd class.
}

// Displays the information about the program.
void AboutUMPTSCmd::doit()
{
     char str1[100];
     char str2[500];

     strcpy(str2,"Name:     Generation of Maximally Parallel Task Systems\n");
     strcpy(str1,"Authors:  Kamalakar Ananthaneni & Dr. M. H. Samadzadeh\n");
     strcat(str2,str1);
     strcpy(str1,"Language: C++\n");
     strcat(str2,str1);
     strcpy(str1,"Toolkits: Motif, Xt Intrinsics and Xlib\n");
     strcat(str2,str1);
     strcpy(str1,"Date:     16 November 1997\n");
     strcat(str2,str1);
     theInfoDialogManager->post(str2,(void*) this);
}

////////////////////////////////////////////////////////////
// DataCmd.h: Displays information about the data files and
//  displays some sample data files.
////////////////////////////////////////////////////////////
```

```
#ifndef DATACMD_H
#define DATACMD_H
#include "Cmd.h"
#include <iostream.h>

class DataCmd : public Cmd {

 protected:

    virtual void doit(); // Implement the pure virtual doit() function in the
                         // Cmd class.
 public:

    DataCmd ( char *, int );
    virtual void execute();// Overrides the execute in Cmd class.
    virtual const char *const className () { return "DataCmd"; }
};
#endif

//////////////////////////////////////////////////////////
// DataCmd.C: Member functions of the DataCmd class.
//////////////////////////////////////////////////////////
#include "DataCmd.h"
#include <stdlib.h>
#include "DialogManager.h"

DataCmd::DataCmd ( char *name, int active ) :
                    Cmd ( name, active )
{
   // EMPTY
}

void DataCmd::execute()
{
 Cmd::execute();
}

// Called from the Cmd::execute function, to display the information
// about data files.
void DataCmd::doit()
{
    char str1[100];
    char str2[2000];

     strcpy(str2,"The Random data files contains random task systems.\n");
     strcpy(str1,"A sample Random data file:        N = 5\n");
     strcat(str2,str1);
     strcpy(str1,"                                  01000\n");
     strcat(str2,str1);
     strcpy(str1,"                                   0111\n");
     strcat(str2,str1);
     strcpy(str1,"                                    011\n");
     strcat(str2,str1);
     strcpy(str1,"                                     01\n");
     strcat(str2,str1);
     strcpy(str1,"                                      0\n\n");
     strcat(str2,str1);

     strcpy(str1,"The Fixed data files contains task systems and their\n");
     strcat(str2,str1);
     strcpy(str1,"associated domains and ranges.\n");
     strcat(str2,str1);
     strcpy(str1,"A sample Fixed data file:         N = 7\n");
     strcat(str2,str1);
     strcpy(str1,"                                  0111000\n");
     strcat(str2,str1);
     strcpy(str1,"                                   000100\n");
     strcat(str2,str1);
     strcpy(str1,"                                    00100\n");
     strcat(str2,str1);
     strcpy(str1,"                                     0010\n");
     strcat(str2,str1);
     strcpy(str1,"                                      010\n");
     strcat(str2,str1);
     strcpy(str1,"                                       01\n");
     strcat(str2,str1);
```

```
    strcpy(str1,"                                          0\n\n");
    strcat(str2,str1);
    strcpy(str1,"                                    6\n");
    strcat(str2,str1);
    strcpy(str1,"                                    0 () 2 (0,1)\n");
    strcat(str2,str1);
    strcpy(str1,"                                    1 (0) 1 (3)\n");
    strcat(str2,str1);
    strcpy(str1,"                                    1 (1) 1 (4)\n");
    strcat(str2,str1);
    strcpy(str1,"                                    2 (1,2) 1 (5)\n");
    strcat(str2,str1);
    strcpy(str1,"                                    1 (2) 0 ()\n");
    strcat(str2,str1);
    strcpy(str1,"                                    2 (0,3) 0 ()\n");
    strcat(str2,str1);
    strcpy(str1,"                                    1 (4) 0 ()\n\n");
    strcat(str2,str1);

    strcpy(str1,"-In the Fixed data file, '6' indicates the number of memory cells.\n");
    strcat(str2,str1);
    strcpy(str1," The next lines indicate the domains and ranges.\n");
    strcat(str2,str1);
    strcpy(str1," For Example, in the line 2 (1,2) 1 (5), \n");
    strcat(str2,str1);
    strcpy(str1," 2 is the number of domain cells and (1,2) is the domain,\n");
    strcat(str2,str1);
    strcpy(str1," 1 is the number of range cells and (5) is the range,\n");
    strcat(str2,str1);
    strcpy(str1," in that order.\n");

    // Post the information in an Information dialog box.
    theInfoDialogManager->post(str2,
    (void*) this,);
}

////////////////////////////////////////////////////////////
// HowToRunCmd.h: Display how to run the application.
////////////////////////////////////////////////////////////
#ifndef HOWTORUNCMD_H
#define HOWTORUNCMD_H
#include "Cmd.h"
#include <iostream.h>

class HowToRunCmd : public Cmd {

 protected:

    virtual void doit();
 public:

    HowToRunCmd ( char *, int );
    virtual void execute();//overrides the execute in Cmd class
    virtual const char *const className () { return "HowToRunCmd"; }
};
#endif

////////////////////////////////////////////////////////////
// HowToRunCmd.C: Provide help about running the tool, and
//  provide information about the various options available.
////////////////////////////////////////////////////////////
#include "HowToRunCmd.h"
#include <stdlib.h>
#include "DialogManager.h"

HowToRunCmd::HowToRunCmd ( char *name, int active ) :
                    Cmd ( name, active )
{
    // EMPTY
}

void HowToRunCmd::execute()
{
    Cmd::execute();
}
```

```
// Display the information about running the algorithm in a dialog box.
void HowToRunCmd::doit()
{
    char str1[100];
    char str2[1000];

     strcpy(str2,"1.In the File menu,\n");
     strcpy(str1," -Select Open, to select an input file.\n");
     strcat(str2,str1);
     strcpy(str1,"  In the File Selection Box that appears, ensure that the \n");
     strcat(str2,str1);
     strcpy(str1,"  directory is /g/akamala/thesis/GUI and select a file \n");
     strcat(str2,str1);
     strcpy(str1,"  with the .dat extension.\n\n");
     strcat(str2,str1);
     strcpy(str1," -Select Run to run the algorithm.\n\n");
     strcat(str2,str1);
     strcpy(str1,"2.In the View menu, select one of the following:\n");
     strcat(str2,str1);
     strcpy(str1," -Select Adjacency Matrix, to view the adjacency\n");
     strcat(str2,str1);
     strcpy(str1,"  matrices of the determinate and the corresponding\n");
     strcat(str2,str1);
     strcpy(str1,"  maximally parallel task systems. \n\n");
     strcat(str2,str1);
     strcpy(str1," -Select Determinate Graph, to view the precedence \n");
     strcat(str2,str1);
     strcpy(str1,"  graph of the determinate task system. \n\n");
     strcat(str2,str1);
     strcpy(str1," -Select Maximally Parallel Graph, to view the precedence \n");
     strcat(str2,str1);
     strcpy(str1,"  graph of the maximally parallel task system \n\n");
     strcat(str2,str1);
     strcpy(str1," -Select Results to view the comparisons between the \n");
     strcat(str2,str1);
     strcpy(str1,"  determinate and the corresponding maximally parallel \n");
     strcat(str2,str1);
     strcpy(str1,"  task systems. \n");
     strcat(str2,str1);


    // Post the information.
    theInfoDialogManager->post(str2,
                               (void*) this,);
}

////////////////////////////////////////////////////////////////////////
// GenerateYourDataCmd.h: Command to provide help on generating data
//  files for the program.
////////////////////////////////////////////////////////////////////////
#ifndef GENERATEYOURDATA_H
#define GENERATEYOURDATA_H
#include "Cmd.h"
#include <iostream.h>

class GenerateYourDataCmd : public Cmd {

 protected:

    virtual void doit(); // Overrides the pure virtual doit() function in the
                         // Cmd class.
 public:

    GenerateYourDataCmd ( char *, int );
    virtual void execute();//overrides the execute in Cmd class
    virtual const char *const className () { return "GenerateYourDataCmd"; }
};
#endif

////////////////////////////////////////////////////////////////////////
// GenerateYourDataCmd.C: This command displays information regarding
//  creating input files to the program.
////////////////////////////////////////////////////////////////////////
#include "GenerateYourDataCmd.h"
#include <stdlib.h>
#include "DialogManager.h"
```

```
GenerateYourDataCmd::GenerateYourDataCmd ( char *name, int active ) :
                    Cmd ( name, active )
{
   // EMPTY
}

void GenerateYourDataCmd::execute()
{
 Cmd::execute();
}

// Post the information about generating data files, in response to the
// user activating the object.
void GenerateYourDataCmd::doit()
{
    char str1[100];
    char str2[1000];

    strcpy(str2,"To generate a random task system, do the following: \n");
    strcpy(str1,"Type /y/samad/PAPER/KAMALA/task_graph.out \n\n");
    strcat(str2,str1);
    strcpy(str1,"Follow the instructions and give '1' when asked for the \n");
    strcat(str2,str1);
    strcpy(str1,"Number of graphs to be generated and placed in graphs.out. \n\n");
    strcat(str2,str1);
    strcpy(str1,"-A file graphs.out is copied into your current directory \n\n");
    strcat(str2,str1);
    strcpy(str1,"-Copy graphs.out into Random.dat to run the program using \n");
    strcat(str2,str1);
    strcpy(str1," random domains and ranges. \n\n");
    strcat(str2,str1);
    strcpy(str1,"-Copy graphs.out into Fixed.dat if you want to provide your \n");
    strcat(str2,str1);
    strcpy(str1," domains and ranges. \n");
    strcat(str2,str1);
    strcat(str2,str1);
    strcpy(str1,"-Refer to help on Data to look at sample Random and Fixed data files.
                \n");
    strcat(str2,str1);

    theInfoDialogManager->post(str2,
                               (void*) this,);
}

////////////////////////////////////////////////////////////////
// Point.h: Represents a point.
////////////////////////////////////////////////////////////////
#ifndef POINT_H
#define POINT_H
#include <iostream.h>
class Point{
 private:
    int _x;                 // x co-ordinate of the point.
    int _y;                 // y co-ordinate of the point.
 public:
    Point();
    void setXY(int, int);   // set the x and y co-ordinates.
    int x(){return _x;}
    int y(){return _y;}
    void operator=(Point&); // Overloaded assignment operator.
    ~Point(){};
};
#endif;

#include "Point.h"

// Constructor.
Point::Point()
{
    _x=0; _y=0;
}

// Sets the co-ordinates of the point.
void Point::setXY(int x, int y)
{
```

```
    _x = x;
    _y = y;
}

// Overloaded equality operator. Accepts an instance of Point
// as argument and assigns x and y to the current instance of Point.
void Point::operator=(Point& P)
{
    _x = P._x;
    _y = P._y;
}

////////////////////////////////////////////////////////////////////////////
// UsedPoints.h: This class is used to store points in the drawing area widget
//  that are already used. These points are checked with before assigning a
//  a Point to a node, so that no two nodes overlap each other.
////////////////////////////////////////////////////////////////////////////
#ifndef USEDPOINTS_H
#define USEDPOINTS_H
#include "Point.h"
#include <assert.h>

class UsedPoints {

 private:
   Point *_pts;
   int _numOfUsedPoints;

 public:
    UsedPoints(int);
    void add(Point&);
    int contains(Point&);
    void rmPoint(Point&);
    void empty(){_numOfUsedPoints = 0;}
    ~UsedPoints();
};
#endif

////////////////////////////////////////////////////////////////////////////
// UsedPoints.C: Member functions of the UsedPoints class.
////////////////////////////////////////////////////////////////////////////
#include "UsedPoints.h"

UsedPoints::UsedPoints(int NumOfTasks)
{
    _pts = new Point[NumOfTasks];
    assert(_pts);
    _numOfUsedPoints = 0;
}

// This function receives point class instance as an argument and adds it to the
// array of used points.
void UsedPoints::add(Point& P)
{
    _pts[_numOfUsedPoints] = P;
    _numOfUsedPoints++;
}

// This function checks to see if point p exists in the array of used points and if so
// removes it.
void UsedPoints::rmPoint(Point& P)
{
    for(int i=0;i<_numOfUsedPoints;i++)
    if((_pts[i].x() == P.x()) && (_pts[i].y() == P.y())) // Check if it exists/
    {
      for(int k = i; k < _numOfUsedPoints - 1;k++)
      _pts[k] = _pts[k+1];
      break;
    }
    // Decrement the count since one point is removed.
    _numOfUsedPoints--;
}

// Checks if the passed argument P, overlaps with the Points in the array.
// For a node the topLeft Point is stored. So the function makes sure that the
// the square represented by P does not overlap with the square represented by
```

```
// by the points in the UsedPoints array.
int  UsedPoints::contains(Point& P)
{
    for(int i=0;i<_numOfUsedPoints;i++)
    {
      if( _pts[i].x() >= P.x()) // If the x co-ordinates of a used point is
      {           // greater than or equal the x co-ordinate of P,
         if(_pts[i].y() >= P.y()) // check for overlapping of array.
         {
           // If there is difference of less than 50 in x co-ordinates, or a
           // a difference of less than 100 in the y co-ordinates return
           // 1 indicating that an overlap has occurred.
           if(((_pts[i].x()-P.x()) < 50) && ((_pts[i].y() - P.y()) < 100))
            return 1;
         }
         else
         if(P.y() > _pts[i].y())
         {
             if(((_pts[i].x()-P.x()) < 50) && ((P.y() - _pts[i].y()) < 100))
               return 1;
         }
      }
      else
      if( _pts[i].x() < P.x())
      {
         if(_pts[i].y() >= P.y())
         {
             if(((P.x()- _pts[i].x()) < 50) && ((_pts[i].y() - P.y()) < 100))
               return 1;
         }
         else
         if(P.y() > _pts[i].y())
         {
            if(((P.x() - _pts[i].x()) < 50) && ((P.y() - _pts[i].y()) < 100))
               return 1;
         }
        }

    }
    // No overlap occurred.
    return 0;
}

// Destructor.
UsedPoints::~UsedPoints()
{
 delete [] _pts;
}

////////////////////////////////////////////////////////////////////////
// Node.h: This class stores the points in the drawing area for the
//  nodes. Each node represents a task.
////////////////////////////////////////////////////////////////////////

#ifndef NODE_H
#define NODE_H

#define TRUE 1
#define FALSE 0

#include <Xm/Xm.h>
#include <Xm/DrawingA.h>
#include "IntArray.h"
#include "Point.h"

class Node {
 private:

    Point _topLeft;       // Top left point of the square.
    Point _topCenter;     // Top center point of the circle.
    Point _bottomCenter; // Bottom center point in the circle.

    int _set;             // Flag indicating, if the node was previously set.
    char* _taskNumber;

 public:
```

```
    Node();
    void setPoints(Point,int);
    void draw(GC*,Widget*);    // Draws the node.
    int set(){return _set;}
    void reset(){_set = FALSE;}

    // Functions that return the points in the node.
    Point topCenter(){ return _topCenter;}
    Point bottomCenter(){ return _bottomCenter;}
    Point topLeft(){ return _topLeft;}
    ~Node();
};
#endif;

////////////////////////////////////////////////////////////////////////////
// Node.C: The Node class represents a node in the precedence graph.
//  A node in the graph represents a task.
////////////////////////////////////////////////////////////////////////////
#include "Node.h"
#include <string.h>
#include <stdio.h>

// Constructor.
Node::Node()
{
 _taskNumber = new char[3];
 _set = FALSE;
}

// This function sets the co-ordinates of the node.
// The argument P is the topleft point of the square around the arc.
// TaskNum indicates the task number and is used in displaying the
// number in the node.
void Node::setPoints(Point P, int TaskNum)
{
 _set = TRUE; // Indicates that the Node is set.
 _topLeft = P;
 _topCenter.setXY(P.x()+21,P.y()); // Since the diameter of the circle is
                                   // 42 the topCenter's x co-ordinate is obtained by
                                   // adding 21 to the topLeft's x co-ordinate.

 _bottomCenter.setXY(P.x()+21,P.y()+42); // Since the diameter of the circle is 42, the
                                         // bottomCenter's y co-ordinate is obtained by
                                         // adding 21 to topLeft's y-co-ordinate.
 sprintf(_taskNumber, "%c%d", 'T', TaskNum);
}

// The function draws the Node, by drawing the circle and the string
// representing the task number.
void Node::draw(GC* gc, Widget* drawing_area)
{
    XDrawArc(XtDisplay(*drawing_area), XtWindow(*drawing_area),
            *gc, _topLeft.x(), _topLeft.y(), 42, 42, 0, 360*64);

    // Put the string holding the task number, in the arc.
    XDrawImageString(XtDisplay(*drawing_area),XtWindow(*drawing_area),
    *gc, _topLeft.x()+10,_topLeft.y()+25,_taskNumber,strlen(_taskNumber));
}


Node::~Node()
{
 delete  _taskNumber;
 _taskNumber = NULL;
}

////////////////////////////////////////////////////////////////////////
// line.h: Represents a line. A line is drawn between two nodes
//  indicating dependency between the tasks represented by the nodes.
////////////////////////////////////////////////////////////////////////
#ifndef LINE_H
#define LINE_H
#include <Xm/Xm.h>
#include <Xm/DrawingA.h>
#include "Point.h"
```

```
class line
{
 private:
 Point _lineStart;  // Starting point of the line. The bottom center of
                    // a node.
    Point _lineEnd; // Ending point of a line. The top center of a node.
 public:
    line(){};
    void setPoints(Point&, Point&);
    void draw(GC*,Widget*);
};
#endif;

////////////////////////////////////////////////////////////////////////////////
// line.C: Contains member functions of the line class, used to set the co-ordinates
//  of the line and to draw the line in a drawing area widget.
////////////////////////////////////////////////////////////////////////////////
#include "line.h"

// Set the starting and ending points of the line.
void line::setPoints(Point& lineStart, Point& lineEnd)
{
    _lineStart = lineStart;
    _lineEnd = lineEnd;
}

// This function draws the line. The graphics context and the drawing area
// widget into which it needs to be drawn are passed to the function.
void line::draw(GC *gc, Widget *drawing_area)
{
    XDrawLine(XtDisplay(*drawing_area),XtWindow(*drawing_area),
                      *gc, _lineStart.x(), _lineStart.y(), _lineEnd.x(), _lineEnd.y());
}

////////////////////////////////////////////////////////////////////////////////
// UMPTSApp.h: Application class for the UMPTS generation. This class is derived
//  from the application class.
////////////////////////////////////////////////////////////////////////////////
#ifndef MENUDEMOAPP_H
#define MENUDEMOAPP_H
#include "Application.h"
#include "TaskGraph.h"
#include <string.h>
#include "head.h"
class ResultsWindow;
class UMPTSApp : public Application {

 protected:
    TaskGraph *_determinate;
    TaskGraph *_maximallyPll;

    char *_fullFileName; // Indicates the full path name of the file.
    char *_fileName;     // Indicates the name of the file.
 public:
    friend class ResultsWindow;
    friend class MatrixWindow;
    UMPTSApp ( char * );
    ~UMPTSApp();

    void setFileName(char*, char*);
    int setMemory(ifstream&);

    int runAlgorithm(); // Runs the algorithm.
    void draw(TaskSysEnum);
    virtual const char *const className() { return "UMPTSApp"; }
};

extern UMPTSApp *theUMPTSApp;
#endif

////////////////////////////////////////////////////////////////////////////////
// UMPTSApp.C: This file contains the member functions of the UMPTSApp class.
////////////////////////////////////////////////////////////////////////////////
#include "UMPTSApp.h"
#include "MenuWindow.h"
#include "RandGen.h"
```

```
#include "DialogManager.h"
#include <assert.h>

UMPTSApp *theUMPTSApp = new UMPTSApp ( "UMPTS algo" );
MainWindow  *theMainWindow =  new MenuWindow ( "Generation of Maximally Parallel Task
Systems" );

UMPTSApp::UMPTSApp ( char * name ) : Application ( name )
{
    _determinate = NULL;
    _maximallyPll = NULL;
}

void UMPTSApp::setFileName(char* fullFileName, char* fileName)
{
    _fullFileName = new char[strlen(fullFileName)+1];
    _fileName = new char[strlen(fileName)+1];
    strcpy(_fileName, fileName);
    strcpy(_fullFileName, fullFileName);


}

// This function sets the memory by passing the exact input file name
// to the setMemory function in the TaskSystem class.
int UMPTSApp::setMemory(ifstream& fin)
{
     RandGen Rand;
     int limit = 100;
     if(strspn(_fileName, "Fixed")==5)     // Check if the file name has a prefix 'Fixed'.
     {
        if(!_determinate->setMemory(fin))
         return 0;
     }
     else
     if(strspn(_fileName, "Random")==6)    // Check if the file name has a prefix 'Random'.
         _determinate->setMemory(Rand, limit);

     return 1;
}

// This function draws the graphs by calling the draw() function in the TaskGraph class.
void UMPTSApp::draw(TaskSysEnum TASKSYSTEM)
{
    char str[100];
    if(_determinate->numOfTasks() > 20)
    {
        strcpy(str, "Cannot draw Graph. Too many tasks. \n");
        strcat(str, "Graph may be too cluttered to view.");
        theInfoDialogManager->post(str,(void *) this);
    }
    else
    {
       if(TASKSYSTEM == DETERMINATE)
     _   _determinate->draw();
       else
        if(TASKSYSTEM == MAXPLL)
          _maximallyPll->draw();
    }
}

// This function implements the UMPTS algorithm by calling the appropriate functions.
int UMPTSApp::runAlgorithm()
{
    int temp=0;
    ifstream f_in(_fullFileName);
    int numOfTasks;

    // Allocate memory for the task systems.
    _determinate = new TaskGraph("Determinate Task System");
    assert(_determinate);
    _maximallyPll = new TaskGraph("Maximally Parallel Task System");
    assert(_maximallyPll);

    // Read the input task system into the determinate task system.
    f_in >> _determinate;
```

```
    numOfTasks = _determinate->numOfTasks();

    // If input file format is wrong, then return.
    if(!UMPTSApp::setMemory(f_in))
    return 0;                          // 0 indicates an error has occured.

    _determinate->makeDeterminate();  // Makes the task system determinate.

    // Removes the redundant dependencies.
    _determinate->checkRedundancy();

    // Obtain the degree of parallelism of the task system.
    _determinate->degOfParallelism();

    _determinate->setLevel();        // Sets the levels of the tasks in the task system.

    // Copy the determinate task system into the maximally parallel task system.
    (TaskSystem)*_maximallyPll = (TaskSystem)*_determinate;

    // Obtains the maximal parallelism in the task system by placing dependencies
    // between mutually interfering tasks and removing all other dependencies.
    _maximallyPll->getMaxParallelism();

    // Remove the redundancy in the maximally parallel task system.
    _maximallyPll->checkRedundancy();

    // Obtain the degree of parallelism for the maximally parallel task system.
    _maximallyPll->degOfParallelism();

    _maximallyPll->setLevel(); // Set the levels of the tasks in the maximally parallel
                               // task system.
    f_in.close();

    return 1; // 1 indicates the algorithm was run successfully.
}

// Destructor.
UMPTSApp::~UMPTSApp()
{
    delete [] _determinate;
    delete [] _maximallyPll;
    _determinate = NULL;
    _maximallyPll = NULL;
}

////////////////////////////////////////////////////////////////////////
// Canvas.h: An abstract class derived from MainWindow class.
//  It has a drawing area widget and a pushButton as its data members.
////////////////////////////////////////////////////////////////////////
#ifndef CANVAS_H
#define CANVAS_H

#include <Xm/Xm.h>
#include <Xm/DrawingA.h>
#include <Xm/MainW.h>
#include "Node.h"
#include "line.h"
#include "MainWindow.h"

class Canvas: public MainWindow {

 protected:
    Widget _drawingArea; // Drawing area widget.
    Widget _okButton;    // Push button widget.
    GC _gc;

   Node *_node; // Pointers to the Node and the line classes, representing
    line *_line; // the tasks and the dependencies between them.

    // Callback function to be called when the ok button is pressed.
    static void okCallback(Widget, XtPointer, XtPointer);
    void okExecute();

public:
```

```
    Canvas(char *name);
    virtual void initialize(); // Initialize the main window and it's children.

    // Pure virtual functions to be implemented
    // in the derived classes
    virtual void setPoints()=0;
    virtual void draw()=0;

    // Destroys the main window, drawing area and the OK button after the OK button is
    // pressed.
    void unmanage();
    virtual ~Canvas();
};
#endif;

////////////////////////////////////////////////////////////////////////////////////////
// Canvas.C: Initializes the drawing area, main window, and the graphics context and also
//  provides functions for managing and unmanaging them.
////////////////////////////////////////////////////////////////////////////////////////
#include "Canvas.h"
#include <Xm/PushB.h>

// Constructor for the Canvas class.
Canvas::Canvas(char* name):MainWindow(name)
{
    _line = NULL;
    _node = NULL;

}

////////////////////////////////////////////////////////////////////////////////
// The initialize() function creates the drawing area widget and the graphics
// context. The TaskGraph class draws into this drawing area widget.
////////////////////////////////////////////////////////////////////////////////
void Canvas::initialize()
{
    int foreground=0,background=0;
    XGCValues vals;

    // Initialize the main window.
    MainWindow::initialize();

    // Create the drawing area widget and the pushButton widgets as the
    // children of the main window.
    _drawingArea = XmCreateDrawingArea(_main,"drawingArea",0,0);
    _okButton = XtVaCreateWidget("OK",
                                 xmPushButtonWidgetClass, _main,
                                 NULL);

    XtAddCallback(_okButton,
                  XmNactivateCallback,&Canvas::okCallback,
                  (XtPointer)this);

    // Get the values for foreground and background for the drawing area widget.
    XtVaGetValues(_drawingArea,
                  XmNforeground, &foreground,
                  XmNbackground, &background,
                  NULL);

    // Get the Graphics Context needed to draw into the drawing area widget.
    vals.foreground = foreground;
    vals.background = background;
    vals.line_width = 1.5;
    _gc = XtGetGC(_drawingArea, GCForeground | GCBackground | GCLineWidth, &vals);

}

// Callback function called when the ok button is activated.
void Canvas::okCallback(Widget, XtPointer clientData, XtPointer)
{
    Canvas *obj = (Canvas*) clientData;
    obj->okExecute();
}

void Canvas::okExecute()
{
```

```
    MainWindow::unmanage();

}

// Make the windows invisible by unmanaging them.
void Canvas::unmanage()
{
    XtUnmanageChild(_drawingArea);
    XtUnmanageChild(_okButton);
    MainWindow::unmanage();
}

// Destructor.
Canvas::~Canvas()
{
    delete [] _line;
    delete [] _node;
    _line = NULL;
    _node = NULL;
}


//////////////////////////////////////////////////////////////////////////////
// TaskGraph.h : Displays the precedence graph of the task system
//  The TaskGraph class is derived from the TaskSystem and the Canvas classes.
//  The Canvas class holds the drawing area widget and the graphics context.
//////////////////////////////////////////////////////////////////////////////
#ifndef TASK_GRAPH_H
#define TASK_GRAPH_H
#include "Canvas.h"
#include "Node.h"
#include "line.h"
#include "TaskSystem.h"
#include "UsedPoints.h"

class TaskGraph: public Canvas, public TaskSystem {
private:
    int _outOfBound;          // Set if the points go outside the drawing area widget.
    UsedPoints *_usedPoints; // Contains a list of points in the drawing area,
                              // that have already been assigned.
                              // Used to make sure that no two nodes intersect.
public:

    TaskGraph(char* name);

    // initialize the window and the widgets.
    void initialize();

    // Draw the graph for the task system.
    void draw();

    // Called when the drawing area widget is exposed.
    void exposeCallback();

    // Set the points for the nodes and the lines.
    void setPoints();
    void setNode(int, int);

    ~TaskGraph(){ delete _usedPoints;}
};
#endif;


#include "TaskGraph.h"
#include "DialogManager.h"
#include <assert.h>
const int OUTOFBOUND = 0;
const int NORMAL  = 1;

void exposeCB(Widget, XtPointer, XtPointer);

TaskGraph::TaskGraph(char *name): TaskSystem(), Canvas(name)
{
    _outOfBound = false; // Initialize to false.
    _usedPoints = NULL;
}
```

```
// This function overrides the initialize() member function in the Canvas class.
// It sets the resources of the widgets and manages them.
void TaskGraph::initialize()
{
    int width =0, height =0;

    //Call the base classes initialize member function
    Canvas::initialize();

    //Allocate memory for the Nodes and lines to be drawn
    _node  = new Node[_numOfTasks];
    _line = new line[_numOfDependencies];
    _usedPoints = new UsedPoints(_numOfTasks);

        assert(_node);
    assert(_line);
    assert(_usedPoints);

    // Dynamically change the width of the main window widget to
    // fit the Precedence Graph.
    width = (_maxDegOfParallelism + 1) * 100 + 200 ;
    height = ( _numOfLevels )*100 + 100;

    // Make sure the height and width of the main window is the same for all the
    // graphs. If the graph size takes more than the size of the main window,
    // set the drawing area widget's size to fit the graph and provide scrollbars
    // to view graphs outside the main window.
    if(width <  960)
    width = 960 ;
    if(height <  550)
    height =550;

    // Set the drawing area widget as the working area for the main window.
    XtVaSetValues(_main,
                  XmNworkWindow, _drawingArea,
                  NULL);

    // Set the resources of the base widget.
    XtVaSetValues(_baseWidget,
                  XmNheight, 650,
                  XmNwidth,999,
                  XmNx,100,
                  XmNdeleteResponse, XmUNMAP,
                  XmNy,200,
                  NULL);

    // Set the resources of the drawing area widget.
    XtVaSetValues(_drawingArea,
                  XmNwidth,  width,
                  XmNheight, height,
                  NULL);

    // Manage the main window and it's children.
    MainWindow::manage();
    XtManageChild(_drawingArea);
    XtManageChild(_okButton);

    // Set the _okButton as the message area for the main window,
    // so that it can be pressed to unmap the window.
    XtVaSetValues(_main,
                  XmNmessageWindow, _okButton, NULL);

    // Set callback so that exposeCB() is called when the drawing area is exposed.
    XtAddCallback(_drawingArea,
                  XmNexposeCallback,
                  exposeCB,
                  (XtPointer) this);
}

// This  function sets the co-ordinates of the nodes . It obtains the width
// and height of the main window widget. Depending on the degree of
// parallelism the co-ordinates of the first node are set. This function
// calls the setXY function to set the Nodes of all the tasks dependent on the
// current task .
void TaskGraph::setPoints()
```

```
{
    int width=0,height=0;
    int x = 0;
    int y = 50;
    Point start, end;
    int lineIndex=0;

    // Get the height and width of the main window.
    XtVaGetValues(_main,
                  XmNwidth, &width,
                  XmNheight, &height,
                  NULL);

    // Reset all the nodes, to indicate their co-ordinates have not yet
    // been set.
    for(int i=0;i<_numOfTasks;i++)
    _node[i].reset();

    x = width/(_numOfLevelZeros + 1);

    // Making sure the starting node is not too much to the left.
    if(x<200)
    x = 200;

    // Set the co-ordinates for the nodes.
    for( i=0;i<_numOfTasks;i++)
    {
       // Check if the co-ordinates of the node were already set in previous calls
       // to setXY.
       if(_node[i].set() == false)
       {
          setNode(x, i);

          // If a node's co-ordinates are outside the drawing area widget, return
          // and display a message.
          if(_outOfBound == true)
           return;
          else
          x+= 110; // Nodes at the same level are separated by 110.
       }
    }

    // Set the co-ordinates for the lines . A line indicates a dependency.
    for(i=0;i<_numOfTasks;i++)
    {
        start = _node[i].bottomCenter(); //start indicates the starting point of the line

        for( int j=0; j < _dependency[i]._dependents.size();j++)
        {
             // end is the end-point of the line.
             end = _node[_dependency[i]._dependents[j]].topCenter();

             // Set the co-ordinates of the line by passing, the start and the end points.
             _line[lineIndex].setPoints(start, end);
             lineIndex++;
        }
    }
}

// This function is called from the setPoints member function. This recursively
// calls itself to set the co-ordinates of the Nodes representing tasks
// dependent on Task TaskNum.
void TaskGraph::setNode(int x, int TaskNum)
{
        Point P1;
        int width;
        int TaskNumD; //TaskNumD indicates the task number of the dependent
                      // of a task.
        int y; // Indicate the y co-ordinate.

        // Set the y co-ordinates, based on the level of the task.
        y = _task[TaskNum].level() * 100 + 50;

        if(_outOfBound == true)
        return;
```

```
        // Here num indicates the number of dependents of task TaskNum.
        int num = _dependency[TaskNum]._dependents.size();

        P1.setXY(x, y); // Set a temporary point.

        // Set the co-ordinates of the Node representing the Task.
        while(_usedPoints->contains(P1)) // If the (Point) area representing the circle
        {                                // was already assigned to another node, get
                                         // another free point.
         P1.setXY(P1.x()+50,P1.y());
         x+=50;
        }

        // Add the recently assigned point to the _usedPoints.
        _usedPoints->add(P1);
        _node[TaskNum].setPoints(P1,TaskNum);

        // If the level is odd and the number of dependents is greater than 1,
        // have a different sequence of x values so that lines do not overlap
        // that often. For example if a line is drawn from a Node in level 0
        // to a node in level1 and another is drawn from a Node in level 0 to
        // a node in level 2, this helps reduce the chances of the lines overlapping.
        if((_task[TaskNum].level()%2 == 1) && (num > 1))
        x+=30;

        // Here num is the number of dependents.
        // x is the value of the x co-ordinate of the first dependent.
        // For example if x is 300 and the task has 2 dependents, then the value
        // of x for the first dependent becomes 250.
        x -= ((num - 1) * 50);

        XtVaGetValues(_drawingArea,
                      XmNwidth, &width,
                      NULL);

        if(( x < 0) || (x > width))
        {
          _outOfBound = true;
          return;
        }

        for(int j=0;j<num;j++)
        {
            TaskNumD =_dependency[TaskNum]._dependents[j];

            if(_node[TaskNumD].set() == false)
             setNode(x, TaskNumD);

           // Increment by different values for the nodes at even levels and nodes at
           // odd levels.
           if(_task[TaskNumD].level()%2)
            x+=90;
           else
            x +=100;
         }
}


////////////////////////////////////////////////////////////////////////
// This function calls the draw() functions in the Node and line classes
// to draw the nodes. It passes a pointer to the graphics context and
// drawing area widget to these functions.
////////////////////////////////////////////////////////////////////////
void TaskGraph::draw()
{
 TaskGraph::initialize();
}

// Callback function called when the drawing area is exposed.
void exposeCB(Widget ,XtPointer client_data, XtPointer )
{

    TaskGraph *obj = (TaskGraph *) client_data;
    obj->exposeCallback();
}
```

```
// Clear the window, set the points for the nodes and lines and then draw
// the graph.
void TaskGraph::exposeCallback()
{
     char str[50];

     XClearWindow(XtDisplay(_drawingArea),XtWindow(_drawingArea));

     // Empty the _usedPoints array.
     _usedPoints->empty();

     // Set the co-ordinates of the nodes.
     TaskGraph::setPoints();

     // If the points are outside the drawing area, display a message saying
     // that the graphs cannot be drawn.
     if(_outOfBound == true)
     {
         Canvas::unmanage();
         strcpy(str, "Cannot draw Graph \n");
         strcat(str, "Co-ordinates out of screen.");
         theInfoDialogManager->post(str,(void *) this);
     }
     else
     {
         // Draw the nodes.
         for(int i=0;i<_numOfTasks;i++)
         _node[i].draw(&_gc, &_drawingArea);

         // Draw the lines.
         for(i=0;i<_numOfDependencies;i++)
          _line[i].draw(&_gc, &_drawingArea);
     };
}

//////////////////////////////////////////////////////////////////////
// TaskSystem.h: The TaskSystem class is the heart of the program.
//  It contains all the tasks in the task system and dependencies
//  between them. The tasks and dependencies are IntArray instances.
//////////////////////////////////////////////////////////////////////

#ifndef TASKSYS_H
#define TASKSYS_H
#include "Task.h"
#include "Dependency.h"
#include <assert.h>
class ResultsWindow;

class TaskSystem{

 protected:

    TaskClass* _task; // Tasks in the task system.
    Dependency* _dependency; // Dependencies between the tasks.

    int _numOfTasks;
    char **_adjacency; // Adjacency Matrix.
    int _numOfMemCells;
    int _maxDegOfParallelism;
    int _numOfDependencies;   / Number of dependencies in the task system.
    int _numRdntDependencies; // Number of redundant dependencies.
    int _numOfLevels; // Number of levels in the task system.
    int _numOfLevelZeros; // Number of root nodes.
 public:
    friend class ResultsWindow;
    friend class MatrixWindow;
    TaskSystem();
    void setNumOfTasks(int);
    void setMemory(RandGen&, int); // Randomly sets the upper limit on memory
    int setMemory(ifstream&);      // Set the domains and ranges for the task
                                   // from an input file.

    // The checkRedundancy() and rmRedundancy() functions are the two
    // functions used to remove redundancy in the task system.
    void checkRedundancy();        // It calls the rmredundancy() function to remove
                                   // redundancy.
```

```
    int rmRedundancy(int, int, int);

    // Obtain redundant dependencies.
    void AddRedundDpOn(int, IntArray*);
    void AddRedundDpts(int, IntArray*);

    void makeDeterminate();     // Make the task system determinate.

    void getMaxParallelism();  // Obtains the Maximum parallelism in the task system.
    int numOfTasks(){ return _numOfTasks;}

    void degOfParallelism();            // Compute the degree of parallelism of the task
    int getDegOfParallelism(IntArray&); // system.

    // Set the levels of the tasks in the task system.
    void setLevel();
    void setSubTreeLevel(int, int);

    // Overloaded input operator.
    friend ifstream& operator>>(ifstream&, TaskSystem*);
    TaskSystem& operator=(const TaskSystem&);
    ~TaskSystem();
};
#endif;

////////////////////////////////////////////////////////////////////////////
// TaskSystem.C: This file contains the member functions for the TaskSystem
// class.
////////////////////////////////////////////////////////////////////////////

#include "TaskSystem.h"
#include <ctype.h>
#include <string.h>
#include <stdio.h>

const int REMOVE = 1;
const int DONTREMOVE =  0;
const int REMOVED =  1;

// Global arrays in the class, holding the complete dependencies  of every
// task in the task system. e.g., RdDependents[0] contains the
// task numbers of all the tasks which depend on the task '0', whether
// directly or indirectly. RdDependsOn[3] contains the task numbers of all
// tasks on which task '3' depends, whether directly or indirectly.
static IntArray *RdDependsOn;
static IntArray *RdDependents;

// RdDependency is obtained by adding RdDependsOn and RdDependents.
// e.g., RdDependency[3] contains all the task that depend on task '3'
// and all the tasks that task '3' depends on (both direct and indirect).
static IntArray *RdDependency;
static IntArray *checkedWith;

TaskSystem::TaskSystem()
{
 _task = NULL;
 _dependency = NULL;
 _adjacency = NULL;
 _maxDegOfParallelism = 0;
 _numOfTasks = 0;
 _numOfMemCells = 0;
 _numOfDependencies=0;
 _numRdntDependencies=0;
 _numOfLevels = 0;
 _numOfLevelZeros = 0;
}

// Sets memory by calling the random number generator.
void TaskSystem::setMemory(RandGen& Rand, int numOfMemCells)
{
 if(_numOfTasks==0)
 return;

 _numOfMemCells = numOfMemCells;
  for(int i=0;i<_numOfTasks;i++)
         _task[i].setDomRngs(Rand, _numOfMemCells);
```

```
}
// Sets memory by reading from the input file. If successful in reading,
// return 1. If error occurred, return 0.
int TaskSystem::setMemory(ifstream& fin)
{
    char ch;

    ch = fin.peek(); // get the character from the file without removing it.

    // Remove leading spaces and new lines from the file stream.
    while((ch == ' ') || (ch == '\n'))
    {
      fin.get(ch);
      ch = fin.peek();
    }

    if(isdigit(ch))
    fin >> _numOfMemCells;
    else
    {
     return 0;
    }

    ch = fin.peek();

    // Skip newline and space characters.
    while((ch == '\n') || (ch == ' '))
    {
       fin.get(ch);
       ch = fin.peek();
    }

    for(int i=0;i<_numOfTasks;i++)
    {
       // If setting domains and ranges failed, return a 0.
       if(!_task[i].setDomRngs(fin, _numOfMemCells))
       return 0;

       if(fin.eof()) // If end of file is eached and the domains an ranges of all
       return 0;     // the tasks have not yet been set, return 0.
    }

    // If domains and ranges were set successfully return 1.
    return 1;
}


// This function obtains the maximum parallelism from the task
// system by checking for the Bernstein's conditions.
void TaskSystem::getMaxParallelism()
{
    int temp1,temp2,temp3;


    for(int i=0;i<_numOfTasks-1;i++)
    {

        for(int j=i+1;j<_numOfTasks;j++)
        {
         // Check for mutually interfering tasks by checking for
         // Bernstein's conditions.
         // Here the operator & stands for "Intersection".
         temp1 = _task[i]._taskMem._domain & _task[j]._taskMem._range;
         temp2 = _task[j]._taskMem._domain & _task[i]._taskMem._range;
         temp3 = _task[i]._taskMem._range & _task[j]._taskMem._range;

         // If dependency does not exist between mutually interfering tasks,
         // place a dependency between them.
         if((temp1 | temp2 | temp3) == 1)
         {
                if(_adjacency[i][j] == '0')
                {
                 _adjacency[i][j]='1';
                 _dependency[i]._dependents.add(j);
                 _dependency[j]._dependsOn.add(i);
                }
```

```
        }
      }
    }
}

// This function adds the tasks on which the tasks in the array
// Arr depends to the TempArray.
// It recursively calls itself and adds all the tasks on which
// the tasks in Arr depends to TempArray.
void AddToTempArray(IntArray& Arr, IntArray* TempArray)
{
    IntArray Array(TempArray->SIZE());

    for(int i=0;i<Arr.size();i++)
        Array = (Array + RdDependency[Arr[i]]);

    Array = (Array - *TempArray); // Remove tasks that have already been accounted
                                  // for to save time.
    //Check if all the tasks have been accounted for.
    if(Array.size() > 0)
    {
       *TempArray = (*TempArray + Array);
       AddToTempArray(Array, TempArray);
    }
}

// This function computes and returns the maximum degree of parallelism
// in the task system. It first obtains the redundant dependencies between
// the tasks and stores then in RdDependsOn and RdDependents arrays. The
// RdDependency array is obtained by adding RdDependsOn and RdDependents
// arrays. Each group of tasks that have direct or indirect dependencies
// are grouped in the TempArray and their degree of parallelism is computed.
// The maximum degree of parallelism is obtained by adding the degree of
// parallelism in all such dependent groups. The tasks in a group have
// dependencies within them but, have no dependencies with tasks in other groups.
void TaskSystem::degOfParallelism()
{
    IntArray *TasksChecked; // Array used to prevent repetetive checking.
    IntArray *TempArray;
    IntArray Array(_numOfTasks);
    int temp; // Temporarily holds the degree of parallelism of TempArray of tasks.

    // If there are no tasks the degree of parallelism is 0 and if the number
    // of tasks  is 1 the degree of parallelism is 1
    if(_numOfTasks == 0)
     return;
    else
    if(_numOfTasks==1)
    {
     _maxDegOfParallelism = 1;
     return;
    }

    _numRdntDependencies = 0; // Reset.

    assert(TasksChecked = new IntArray(_numOfTasks));
    assert(RdDependency = new IntArray[_numOfTasks]);
    assert(RdDependsOn = new IntArray[_numOfTasks]);
    assert(RdDependents = new IntArray[_numOfTasks]);

    _maxDegOfParallelism = 0; // Reset.

    for(int i=0;i<_numOfTasks;i++)
    {
        RdDependency[i].setArr(_numOfTasks);
        RdDependsOn[i].setArr(_numOfTasks);
        RdDependents[i].setArr(_numOfTasks);
    }

    // Form the RdDependsOn array for each task.
    for(i=_numOfTasks-1;i>=0;i--)
     AddRedundDpOn(i, TasksChecked);

    //Empty the tasks checked array
    TasksChecked->empty();
```

```
    // Fill the RdDependents array for each task.
    for(i=0;i<_numOfTasks;i++)
      AddRedundDpts(i, TasksChecked);

    TasksChecked->empty();

    // Form the RdDependency array by adding the RdDependsOn and RdDependents arrays.
    for(i=0;i<_numOfTasks;i++)
    {
      RdDependency[i]= RdDependsOn[i] + RdDependents[i];
      _numRdntDependencies += RdDependents[i].size(); // Obtain the number of redundant
     }                                                // dependencies.
    // Temp Array contains the numbers of tasks dependent on the current task.
    assert(TempArray = new IntArray(_numOfTasks));

    // For each task make the TempArray from it's dependents and calculate
    // the degree of parallelism
    for(i=0;i<_numOfTasks;i++)
    {
     if(TasksChecked->notContains(i))
     {
        *TempArray = RdDependents[i];
        for(int j=0;j<RdDependents[i].size();j++)
        {
         temp = RdDependents[i][j];

         // Check if task with the Id temp depends on any task
         // other than the tasks in TempArray. If so add them to
         // TempArray.
          Array = (RdDependsOn[temp] - *TempArray);
          *TempArray = (*TempArray + RdDependsOn[temp]);
          AddToTempArray(Array, TempArray);
        }

        // Now TempArray contains a complete set of tasks with either
        // direct or indirect dependencies between them.
        // If TempArray is empty it indicates the task i does not have any dependents
        // and hence can be executed in parallel.
        // Otherwise obtain the degree of parallelism by passing TempArray as
        // argument to the getDegOfParallelism function
        if(TempArray->size() == 0)
                temp=1;
        else
                temp = getDegOfParallelism(*TempArray); // Get the degree of parallelism
                                                        // among the tasks in TempArray, and add it to
        _maxDegOfParallelism +=temp; // the total degree of parallelism.

        // Add the TempArray to Taskschecked so that only tasks that are not
        // checked yet can be checked
        *TasksChecked =(*TasksChecked + *TempArray );
        TempArray->empty();
      }
    }
     delete [] RdDependency;
     delete [] RdDependsOn;
     delete [] RdDependents;
     delete TasksChecked;
     delete [] TempArray;
     RdDependency = NULL;
     RdDependsOn = NULL;
     RdDependents = NULL;
     TasksChecked = NULL;
     TempArray = NULL;
}

// This function manipulates the RdDependsOn array.
// For. Ex: If task 9 depends on task 8 and task 8
// depends on task 7 then RdDependsOn[9] contains both
// 8 and 7. This  function calls itself recursively to
// accomplish this.
void TaskSystem::AddRedundDpOn(int TaskId, IntArray* TasksChecked)
{
     int TaskNum;
     int Val =0;
```

```
    if(TasksChecked->notContains(TaskId))//Recursively call the function only if
    {                                    //task has not been checked before
        TasksChecked->add(TaskId);
        Val = _dependency[TaskId]._dependsOn.size();
        for(int j=0;j<_dependency[TaskId]._dependsOn.size();j++)
        {
              TaskNum = _dependency[TaskId]._dependsOn[j];
              AddRedundDpOn(_dependency[TaskId]._dependsOn[j],TasksChecked);
              RdDependsOn[TaskId].add(TaskNum);

              for(int k=0;k<RdDependsOn[TaskNum].size();k++)
              RdDependsOn[TaskId].add(RdDependsOn[TaskNum][k]);

       }
    }
}

// This function manipulates the RdDependents array. It
// recursively obtains all the tasks that depend on a given task
// and stores these tasks in the RdDependents array.
// e.g., RdDependents[0] contains the task numbers of all the
// tasks that depend on task 0. If task 2 depends on task 0 and
// task 3 depends on task 2 then both 2 and 3 are contained in
// RdDependents[0].
void TaskSystem::AddRedundDpts(int TaskId, IntArray* TasksChecked)
{
    int TaskNum;

    if(TasksChecked->notContains(TaskId))
    {
     TasksChecked->add(TaskId);

        for(int j=0;j<_dependency[TaskId]._dependents.size();j++)
        {

         // Recursively call itself to obtain redundant dependencies.
         AddRedundDpts(_dependency[TaskId]._dependents[j],TasksChecked);
          TaskNum = _dependency[TaskId]._dependents[j];
          RdDependents[TaskId].add(TaskNum);

          for(int k=0;k<RdDependents[TaskNum].size();k++)
                RdDependents[TaskId].add(RdDependents[TaskNum][k]);
        }
    }
}

// This function recursively obtains the degree of parallelism for the tasks
// passed in the array. In the array it takes each task and makes an array of
// tasks not having any dependency with it and computes the degree of
// parallelism  among them by recursively calling itself. Eventually it calls
// itself with just 2 arguments and from then on returns the degree of
// parallelism to the previous level in recursion and so on.
int TaskSystem::getDegOfParallelism(IntArray& Array)
{
    int DegPllsm=1;// DegOfPllsm gives the maximum degree of parallelism in the
                   // passed array.
     int temp =1;  // temp holds the temporary value for the degree of parallelism

     IntArray *TempArray = NULL; // It temporarily contains the tasks that
                                 // do not have any dependency  with a specified task
                                 // For each task in Array TempArray varies

     //check for dependency between any two tasks. 2 tasks between which no
     // dependency exists are considered to be parallel
     for(int i=0;i<Array.size();i++)
     {
        // SIZE() function returns the allocated size of the array and size()
        // returns  the number of elements in the array
        assert(TempArray = new IntArray(Array.SIZE()));

        temp =1;

        for(int j=0;j<Array.size();j++)
        {
               if((RdDependency[Array[i]].notContains(Array[j])) && i!= j)
               TempArray->add(Array[j]); // Tasks not having dependency with the task
```

```
        }                                   // represented by Array[i].

        // If every task has a dependency with the task, then it can't be executed in
        // in parallel with any other task.
        if(TempArray->size()!=0)
        {
           if(TempArray->size()==1)
              temp+=1;
           else
              temp += getDegOfParallelism(*TempArray); // Recursively call itself.
        }

        if(temp > DegPllsm)
               DegPllsm = temp;

        delete [] TempArray;
        TempArray = NULL;
 }

 // Return the degree of parallelism among the tasks in TempArray.
 return DegPllsm;
}

// This function makes the task system determinate by adding dependencies
// between any pair of tasks not satisfying Bernstein's conditions.
void TaskSystem::makeDeterminate()
{
 int temp1,temp2,temp3;
 for(int i=0;i<_numOfTasks-1;i++)
 {
        for(int j=i+1;j<_numOfTasks;j++)
        {
         // Operator & returns one if the arrays passed to it have
         // one common value
         temp1 = _task[i]._taskMem._domain & _task[j]._taskMem._range;
         temp2 = _task[i]._taskMem._range & _task[j]._taskMem._domain;
         temp3 = _task[i]._taskMem._range & _task[j]._taskMem._range;

         if((temp1 | temp2 | temp3) == 1)
         {
            _adjacency[i][j]='1';
            _dependency[i].addToDependents(j);
            _dependency[j].addToDependsOn(i);
          }
        }
  }
}

// This function is called to remove redundant dependencies from the
// task system. This function inturn calls the rmRedundancy function.
void TaskSystem::checkRedundancy()
{
    int temp;      // Variable used to store the index of the element before removing
                   // redundancy.
    int i, j;

    // checkedWith array is used to remove redundant computations
    // in the rmRedundancy function
    checkedWith = new IntArray(_numOfTasks);

    // Starting from the last task , each task is checked with all the tasks
    // on which it depends
    for( i= _numOfTasks -1 ;i>=0;i--)
    {
      for(j= (_dependency[i]._dependsOn.size()-1);j>=0;j--)
      {
           temp = _dependency[i]._dependsOn[j];

           //Call rmRedundancy to remove the redundant dependencies
           //If some dependencies have been removed, the location of the task
           // indexed by j changes, so get the location by calling getIndex function
           if(rmRedundancy(i, temp, DONTREMOVE) == REMOVED)
            j = _dependency[i]._dependsOn.getIndex(temp);
      }
      checkedWith->empty(); // Empties it by removing all the elements.
    }
```

```
     _numOfDependencies=0;
     for(i=0;i<_numOfTasks - 1;i++)
     for(j=i+1;j<_numOfTasks ;j++)
     if(_adjacency[i][j] == '1') _numOfDependencies++;

    delete [] checkedWith;
}

////////////////////////////////////////////////////////////////////////////
// The rmRedundancy function removes the redundancy between Task1 and Task2.
// If Flag is REMOVE it indicates that dependencies can be removed.
////////////////////////////////////////////////////////////////////////////
int TaskSystem::rmRedundancy(int Task1, int Task2,int Flag)
{
    int temp;
    int temp1=0;
    int retVal=0;

    checkedWith->add(Task2);
    if((_adjacency[Task2][Task1] == '1') && (Flag == REMOVE))
    {
        _adjacency[Task2][Task1] = '0';
        _dependency[Task2]._dependents.rmValue(Task1);
        _dependency[Task1]._dependsOn.rmValue(Task2);
        retVal = REMOVED; //indicates that dependency has been removed
    }

    for(int i = (_dependency[Task2]._dependsOn.size()-1);i>=0;i--)
    {
        if(checkedWith->notContains(_dependency[Task2]._dependsOn[i]))
        {
           temp = _dependency[Task2]._dependsOn[i];
           if(rmRedundancy(Task1,temp,REMOVE) == REMOVED)
           {
             i = _dependency[Task2]._dependsOn.size() - 1;
             retVal = REMOVED;
           }
        }
    }
    return retVal;
}

// Set the levels of all the tasks by recursively descending from
// top to bottom.
void TaskSystem::setLevel()
{
    int level=0;

    for(int i=0;i<_numOfTasks;i++)
    {
        if(!_task[i].levelSet())
        {
         _numOfLevelZeros++;
         setSubTreeLevel(i, level);
        }
    }

    _numOfLevels = _task[0].level();
    for( i=1;i<_numOfTasks;i++)
    if(_task[i].level() > _numOfLevels)
    _numOfLevels = _task[i].level();
}

void TaskSystem::setSubTreeLevel(int TaskNum, int level)
{
    // If level was previously set, set it to the maximum level.
    // For example if a task depends on two tasks, one at level 1 and another
    // at level 2, then it's level is 3 (2+1).
    if(_task[TaskNum].levelSet())
    {
        if(level > _task[TaskNum].level())
         _task[TaskNum].setLevel(level);
    }
    else
        _task[TaskNum].setLevel(level);
```

```
    // Set the levels of the dependents.
    for(int i=0; i < _dependency[TaskNum]._dependents.size();i++)
        setSubTreeLevel(_dependency[TaskNum]._dependents[i],level + 1);
}
// This is overloaded input operator. That reads all the tasks in the task
// system. It calls the setValues member function of each task and sets the
// private members of each task.
ifstream& operator>>(ifstream& in,TaskSystem* taskSys)
{
    char temp1,temp2;
    int NumOfTasks;
    int val;
    char* temp;
    int numOfRng=0, numOfDom=0;

    in >> temp1 >> temp2;
    in >> NumOfTasks;
    taskSys->_numOfTasks = NumOfTasks;
    if(taskSys->_numOfTasks != 0)
    {
       assert(taskSys->_task = new TaskClass[NumOfTasks]);
       assert(taskSys->_dependency = new Dependency[NumOfTasks]);
       assert(taskSys->_adjacency = new char*[NumOfTasks]);

       for(int i=0;i<NumOfTasks;i++)
         taskSys->_adjacency[i] = NULL;

       for(i=0;i<NumOfTasks;i++)
       {
          assert(temp = new char[NumOfTasks+1]);

          in >> temp; //Read the row into temp
          val = (NumOfTasks - strlen(temp));

          assert(taskSys->_adjacency[i] = new char[NumOfTasks + 1]);

          // Convert the upper triangular matrix into a nXn matrix by appending
          // zeroes at the front.
          for(int j=0;j<val;j++)
          taskSys->_adjacency[i][j] = '0';

          taskSys->_adjacency[i][j]='\0';
          strcat(taskSys->_adjacency[i],temp);

          // Set the needed data members in the Task class.
          taskSys->_task[i].setValues(NumOfTasks, i);

          // Set the dependencies for the task in the Dependency class.
          taskSys->_dependency[i].setDependencies(i, NumOfTasks, taskSys->_adjacency);
          delete [] temp;

          temp = NULL;
       }
    }
  return in;
}

// This is an overloaded equality operator for the task system.
TaskSystem& TaskSystem::operator=(const TaskSystem& Determinate)
{
 _numOfTasks = Determinate._numOfTasks;
 if(_numOfTasks!=0)
 {

   // Allocate memory for the data members.
   assert(_task = new TaskClass[_numOfTasks]);
   assert(_dependency = new Dependency[_numOfTasks]);
   assert(_adjacency = new char*[_numOfTasks]);

   for(int i=0;i<_numOfTasks;i++)
   {
        _task[i] = Determinate._task[i]; // Copy the tasks by calling the overloaded
                                         // equality operator in the Task class.

        // Copy the adjacency matrix.
```

```
        assert(_adjacency[i] = new char[_numOfTasks + 1]);
        for(int j=0;j<_numOfTasks;j++)
        _adjacency[i][j] = '0';

        _adjacency[i][j] = '\0';
        _dependency[i].setDependencies(i, _numOfTasks, _adjacency);
    }
   _numOfMemCells = Determinate._numOfMemCells;
  }
   _numOfDependencies = 0;
   return *this;
}

// Destructor for the TaskSystem class.
TaskSystem::~TaskSystem()
{
   delete [] _task;
   delete [] _dependency;
   _task = NULL;
   _dependency = NULL;

   // Delete the memory allocated for the adjacency matrix.
   for(int i=0;i<_numOfTasks;i++)
   {
     _adjacency[i]='\0';
     delete [] _adjacency[i];
     _adjacency[i] = NULL;
   }
    delete [] _adjacency;
   _adjacency = NULL;
}

////////////////////////////////////////////////////////////////////////////////
// Dependency.h: Stores the dependency of the task. It has data members to
//  _dependsOn that specifies the tasks on which the task depends and _dependents
// specifying the tasks which depend on it.
////////////////////////////////////////////////////////////////////////////////
#ifndef DEPENDENCY_H
#define DEPENDENCY_H

#include "IntArray.h"

class TaskSystem;

class Dependency {
 private:
   int _taskId;
   // _dependsOn is an IntArray instance and constains Task numbers of all the
   // tasks that the current task depends on.
   IntArray _dependsOn;
   IntArray _dependents; // _dependents contains the task numbers of all the
   // tasks depending on the current task.

 public:
   friend class TaskGraph;
   friend class TaskSystem;

   Dependency(){};
   int taskId(){return _taskId;}

   // Set the _dependsOn and _dependents data members.
   void setDependencies(int,int,char**);

   void addToDependents(int);
   void addToDependsOn(int);

  // Overloaded equality operator.
   Dependency& operator=(const Dependency&);
   ~Dependency(){};
};
#endif;

////////////////////////////////////////////////////////////////////////////////
// Dependency.C: This file contains the member functions of the Dependency class.
//  This file has functions to set, add, and remove dependencies.
////////////////////////////////////////////////////////////////////////////////
```

```
#include "Dependency.h"

// This function is called from the TaskSystem class.
// TaskId and adjacency matrix and number of tasks is
// passed as arguments to the function and it sets the
// dependencies for the task indicated by TaskId.
void Dependency::setDependencies(int TaskId, int NumOfTasks, char** Adjacency)
{
   int i=0;
   _taskId = TaskId;

   _dependents.setArr(NumOfTasks);
   _dependsOn.setArr(NumOfTasks);

   for(i=TaskId+1;i<NumOfTasks;i++)//for the first task check from 1 to end
   {
    // A '1' in the Adjacency Matrix indicates a dependency.
    // For example if there is a '1' in the 2nd row and 3rd column,
    // it indicates that task 3 is dependent on task 2.
    if(Adjacency[TaskId][i]=='1')
    _dependents.add(i);
   }


   // Check if there is a dependency in the adjacency matrix showing
   // that the current task depends on some other tasks.
   for(i=0;i<TaskId;i++)
   {
   if(Adjacency[i][TaskId]=='1')
   _dependsOn.add(i);
   }
}

// This function adds the TaskId to the _dependents array.
void Dependency::addToDependents(int TaskId)
{
   _dependents.add(TaskId);
}

// This function adds the TaskId to the _dependsOn Array.
void Dependency::addToDependsOn(int TaskId)
{
   _dependsOn.add(TaskId);
}

// Overloaded equality operator.
Dependency& Dependency::operator=(const Dependency& DEPENDENCY)
{
   _taskId = DEPENDENCY._taskId;
   _dependsOn = DEPENDENCY._dependsOn;
   _dependents = DEPENDENCY._dependents;

   return (*this);
}

////////////////////////////////////////////////////////////////////////////
// Task.h: This class represents a task. It contains data members for storing
// taskId, domain and range. It has member functions for adding
// values to the domain and  range arrays.
////////////////////////////////////////////////////////////////////////////
#ifndef TASK_H
#define TASK_H

#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include "RandGen.h"
#include "IntArray.h"
#include "Memory.h"
#include <assert.h>

const int true  = 1;
const int false = 0;
class TaskSystem;
```

```
class TaskClass
{
 private:
     int _taskId;
     int _numOfTasks; // Indicates the number of tasks in the taskSystem.
     Memory _taskMem; // Memory associated with the task.

     int _level;      // Indicates the level of the task in the precedence graph.
     int _levelSet;   // Flag indicating if the level is set.

public:
     friend class TaskSystem;
     TaskClass();

     // Member functions to access the private data members.
     int taskId(){return _taskId;}
     int levelSet(){ return _levelSet;}
     int level(){return _level;}

     void setLevel(int level ){ _level = level; _levelSet = true; }

     void setValues(int, int);   // Set teh number of tasks and the task number.
     void setDomRngs(RandGen&, int);
     int setDomRngs(ifstream&, int);
     TaskClass& operator=(const TaskClass&);
     ~TaskClass(){}
};
#endif;

#include "Task.h"

// Constructor for the TaskClass
TaskClass::TaskClass()
{
   _taskId =0;
   _level = 0;
   _levelSet = false;
}

// This function sets the values of the data members
// of the Task class.
void TaskClass::setValues(int Num, int taskNum)
{
   _taskId = taskNum;
   _numOfTasks = Num;
};

// This function sets the domains and ranges of the task
// by calling the setDomain and setRange functions respectively
// in the MemoryClass.
void TaskClass::setDomRngs(RandGen& Rand, int NumOfMemCells)
{
   _taskMem.setDomain(Rand, NumOfMemCells);
   _taskMem.setRange(Rand, NumOfMemCells);
}

//This function is called when the input is an fixed data file. The domain
// and range are passed to the function from TaskSystem class as character
// strings.
int TaskClass::setDomRngs(ifstream& fin ,int UpperLimit)
{
   // Call the setDomRng function in the Memory class.
   if(!_taskMem.setDomRngs(fin, UpperLimit))
   return 0;
   else
   return 1;
}

// Overloaded = operator.
TaskClass& TaskClass::operator=(const TaskClass& Task)
{
   _taskId = Task._taskId;
   _numOfTasks = Task._numOfTasks;
   _taskMem = Task._taskMem;
   _level = Task._level;
   return * this;
```

```
}

//////////////////////////////////////////////////////////////
// Memory.h: This class contains data members _domain and _range.
//  This class represents the memory associated with a task.
//////////////////////////////////////////////////////////////

#include "IntArray.h"
#include "RandGen.h"
class Memory
{
 private:
    IntArray _domain; // Domain of a task.
    IntArray _range; // Range of a task.

 public:
     friend class TaskSystem;

     Memory(){};
     int setDomRngs(ifstream&, int);
     void setDomain(RandGen&, int);
     void setRange(RandGen&, int);
     void AddToDomain(RandGen&, int, int, int);
     void AddToRange(RandGen&, int, int, int);
     Memory& operator=(const Memory&);
     ~Memory(){};
};

#include "Memory.h"
#include <ctype.h>

// This function sets the domain of the task by generating
// random domains and then calling to AddToDomain function
// to add them to the domain.
void Memory::setDomain(RandGen& Rand, int NumOfMemCells)
{
    int NumOfDomCells = 5;
    int Temp;
    int j=0;


    _domain.setArr(NumOfDomCells); // Allocate memory for the array.

    for(int i=0;i<NumOfDomCells;i++)
    {
        Temp = Rand.rand(1,NumOfMemCells);
        AddToDomain(Rand, Temp, i, NumOfMemCells);
    }
}

// This function adds the passed value to the domain if the
// value is not already present in the domain. If it is already
// present in the array, it generates another random number
// within the needed limit and recursively calls itself until it
// gets a value not in domain.
void Memory::AddToDomain(RandGen& Rand, int Temp, int i, int NumOfMemCells)
{
    int j=0;

    while((Temp != _domain[j]) && (j<i))
    j++;

    if(j<i)
    {
      // Temp is already present in _domain. So get another random number
      // and call AddToDomain again.
      Temp = Rand.rand(1,NumOfMemCells);
      AddToDomain(Rand, Temp, i, NumOfMemCells);
    }
    else
     _domain.add(Temp);
}

// Sets the range of the task.
void Memory::setRange(RandGen& Rand, int NumOfMemCells)
{
```

```
    int Temp;
    int j=0;
    int NumOfRngCells = 5;

    _range.setArr(NumOfRngCells);//Allocate memory for the range array

    for(int i=0;i<NumOfRngCells;i++)
    {
       Temp = Rand.rand(1,NumOfMemCells);
       AddToRange(Rand, Temp, i, NumOfMemCells);
    }
}

// This function adds the passed value to the range. If the passed value is
// already in the range, then the function generates a value and calls
// itself. This goes on until a value not in the range is obtained.
// Temp is the value to be added and i indicates the index of the last
// location in the range.
void Memory::AddToRange(RandGen& Rand, int Temp, int i, int NumOfMemCells)
{
    int j=0;

    // Temp is the value to be added to the _range. Check if it is already
    // present in the _range.
    while((Temp != _range[j]) && (j<i))
        j++;

    if(j<i)
    {
        // If j < i, it indicates that Temp is already present in the array
        // So get another random value between 1 and the NumOfMemCells and
        // add call AddToRange again to add it to the range.
        Temp = Rand.rand(1,NumOfMemCells);
        AddToRange(Rand, Temp, i, NumOfMemCells);
    }
    else
     _range.add(Temp);

}

// This functions skips the blanks in the file stream.
void skipBlanks(ifstream& fin)
{
   char ch;

   ch = fin.peek(); // Get the next character without removing from the file stream.

   // Keep removing all the blank characters from the file stream.
   while(ch == ' ')
   {
    fin.get(ch);
    ch = fin.peek();
   }
}

// This function is used to read the domains and ranges from the file.
// In case of error, 0 is returned.
int ReadToArr(ifstream& fin, IntArray& Arr, int UpperLimit)
{

   int NumOfCells, temp;
   char ch;

   skipBlanks(fin); // Skip all white spaces.

   ch = fin.peek();


   // The number of domains and the number of ranges should be integer.
   if(!isdigit(ch))
     return 0;

   fin >> NumOfCells;

   Arr.setArr(NumOfCells);     // Allocate memory.
```

```
   fin >> ch;

   // Check for opening parenthesis.
   if(ch != '(') return 0;

   // Read the numbers ignoring spaces and return 0 in case of error.
   for(int i=0;i<NumOfCells;i++)
   {
        skipBlanks(fin);
        ch = fin.peek();
        if(isdigit(ch))
        {
         fin >> temp;
         if( (temp < 0) || (temp > UpperLimit))
         return 0;

         Arr.add(temp);
   }

   skipBlanks(fin);

   ch = fin.peek();

   // Check for the separator ',' between values.
   if(ch == ',')
   fin >> ch;
   else
   if(ch != ')')
   return 0;

 }

 // Read the ')' character.
 fin >> ch;
 skipBlanks(fin);
 return 1;
}

// In the case of examples, domains and ranges are passed as character strings from the
// TaskSystem class. This function is used to set the domains and ranges for the examples.
int Memory::setDomRngs(ifstream& fin, int UpperLimit)
{
       char ch;

  if(!ReadToArr(fin, _domain, UpperLimit))
        return 0;

  if(!ReadToArr(fin, _range, UpperLimit))
       return 0;

  if(!fin.eof())
  {
       ch = fin.peek();
       if( ch == '\n')
        fin.get(ch);
       else
        return 0;
  }
  ch = fin.peek();

   while((ch == ' ') || (ch == '\n'))
   {
        fin.get(ch);
          ch = fin.peek();
   }

   return 1;
}

// Overloaded = operator for the Memory class.
Memory& Memory::operator=(const Memory& Mem)
{
   _range = Mem._range;
   _domain = Mem._domain;
   return *this;
}
```

```
//////////////////////////////////////////////////////////////////
// RandGen.h: This class represents a random number generator
// Source: "Random Number Generators: Good Ones are Hard to Find",
// by S. K. Park and K. W. Miller, Communications of the ACM,
// Vol. 31, No. 10, October 1988, pp. 1192-1201.
//////////////////////////////////////////////////////////////////

#ifndef RANDOM
#define RANDOM
#include <iostream.h>
#include "math.h"
#include "time.h"

#define a 16807.0
#define m 2147483647.0
#define q 127773.0
#define r 2836.0

class RandGen{
 private:

    double _seed;     // The seed for the generator
    int _low, _high; // _low indicates the lower limit and _high
                     // indicates the UpperLimit. So random numbers are
                     // generated in these limits.

 public:
    RandGen();    // Default constructor.
    int rand(int&, int&); // Takes 2 arguments and sets the upper and
                          // lower limits with them.
    int rand();
    void setHigh(int val){_high = val;}
    double random();
    int normalize();
    ~RandGen(){}
};
#endif;

//////////////////////////////////////////////////////////////////////////////////////////
// RandGen.C: This file contains functions needed to implement a random number generator.
//////////////////////////////////////////////////////////////////////////////////////////

#include "RandGen.h"
#include <unistd.h>

// This is the constructor for the random number generator.
// It uses the time function to initialize the seed.
RandGen::RandGen()
{
 struct tm *ptr;
 time_t Lt;
 _seed =0;

 while(_seed == 0)
 {
    Lt = time(NULL);
    ptr = localtime(&Lt);

    _seed =(double)ptr->tm_sec/60.00;
    if(_seed == 0)
        sleep(1);
 }

  _low = 1;
  _high = 100;
}

// This function generates and returns a random number within the lower and upper limits.
double RandGen::random()
{
 double rand, lo, hi, test;
 int    tmp_int;

 // generate a random number
```

```
 tmp_int = (int)(_seed/q);
 hi      = tmp_int * 1.0;
 lo      = _seed - q * hi;
 test    = a *lo - r *hi;

 if(test > 0.0)
        _seed = test;
 else
        _seed = test + m;

 rand = _seed/m;

return(rand);

}

//This function normalizes the random number generated by the call to
//random() function.
int RandGen::normalize()
{
 long N,
      temp,
      norm;
 double temp2;

 random(); // Make one idle call to random.

 temp2 = -log(random())/2.3890;// 2.3890 is lambda.

 while(temp2 > 1)
               temp2 = temp2 - 1.0;

 temp = m*temp2;
 N = temp %_high;

 if(N == 0)
    norm = _high;
 else if (N < _low)
    norm = _low;
 else
    norm = N;

  return norm;
}

// Returns a random number between LOW and HIGH.
int RandGen::rand(int& LOW, int& HIGH)
{
  _low = LOW;
  _high = HIGH;
  return(normalize());
}

// This function is called to generate a random number .
// This inturn calls the random and normalize functions
// and returns the random number.
int RandGen::rand()
{
  random();
  return(normalize());
}

////////////////////////////////////////////////////////////////////////////
// IntArray.h: This class represents an integer array. It has all the needed
//  overloaded operators to manipulate the array.
////////////////////////////////////////////////////////////////////////////
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream.h>
#include <fstream.h>
#include <assert.h>

class IntArray
{
  int _size; // Size of the array, i.e., the memory allocated.
  int* _arr;
```

```
  int _numOfElements; // Number of integers stored in the array.

 public:
   IntArray();     // Default constructor.
   IntArray(int);  // One argument constructor.
   int SIZE(){ return _size;}
   int size() const {return _numOfElements;}  // Return the number of elements.

   void setArr(int); // Allocates memory for an integer array with the size as
                     // specified by the argument it receives.

   void empty();
   void add(int);         // Function to add a value to the array.
   int& operator[](int); // Overloaded '&' operator.
   void rmValue(int);     // Removes the specified argument from the array if present.
                          // if it is present in it.
   int notContains(int);
   int getIndex(int);

   // Overloaded operators to make array operations simple.
   friend IntArray operator-(IntArray, IntArray);          // overloaded '-' operator.
   friend IntArray operator+(IntArray&, IntArray&);        // '+' operator.
   friend ofstream& operator<<(ofstream&,const IntArray&);// Output operator.
   friend int  operator&(const IntArray& _arr1,const IntArray& _arr2);
   IntArray& operator=(const IntArray& _arr2);             // Assignment operator
   ~IntArray();.
};
#endif;

#include "IntArray.h"
#include <string.h>
#include <stdlib.h>

// Default Constructor.
IntArray::IntArray()
{
   _arr = NULL;
   _size=0;
   _numOfElements=0;
}

// Single argument constructor.
IntArray::IntArray(int size)
{
   _arr = NULL;
   assert(_arr = new int[size]);

   // Initialize all the locations in the array to -1.
   for(int i=0;i<size;i++)
   _arr[i]=-1;
   _size = size;
   _numOfElements=0;
}


// Allocate an integer array with size specified by the argument passed.
void IntArray::setArr(int size)
{
   if(_arr !=NULL)
   delete [] _arr;
   _arr = NULL;

   assert(_arr = new int[size]);
   for(int i=0;i<size;i++) // Initialize all the elements to -1.
   _arr[i]=-1;             // This helps in debugging.

   _size = size;
   _numOfElements=0;
}

//This function empties the array by making the number
// of elements as 0 and then filling the array with -1.
void IntArray::empty()
{
   _numOfElements = 0;
   for(int i=0;i<_size;i++)
```

```
   _arr[i]=-1;
}

// Overloaded output operator to output the arrays contents to a file.
ofstream& operator<<(ofstream& FOUT, const IntArray& Array)
{
   for(int i=0;i<Array._numOfElements;i++)
   FOUT << Array._arr[i] << ' ';
   FOUT << endl;
   return FOUT;
}

// This function takes an integer as argument and adds it to the
// _arr data member representing the integer array.
void IntArray::add(int Val)
{
 int temp;

   for(int i=0;i<_numOfElements;i++)
   {
        if(_arr[i] == Val) // Return if the array already contains Val.
          return;
        else
        if(_arr[i] > Val)
          break;
   }

   // Put the Val in the array.
   for(int j =i; j<_numOfElements;j++)
   {
     temp =_arr[j];
     _arr[j]=Val;
      Val = temp;
   }
   _arr[j]=Val;
   _numOfElements++;
}

// This function accepts an integer as an argument. It checks to see if the
// integer is present in the array. If so it is removed from the array.
void IntArray::rmValue(int taskNum)
{
    int i=0;

    while((_arr[i] < taskNum) && (i<_numOfElements))
    i++;

    if(_arr[i] == taskNum)
    {
        while(i<_numOfElements  - 1) // Move all the elements forward.
        {
           _arr[i]=_arr[i+1];
           i++;
        }
       _arr[i]=-1;
       _numOfElements--;            // Decrease the number of elements in the array.
   }
}

// This function returns the index of the integer passed to it.
// If the integer is not present, it returns a -1.
int IntArray::getIndex(int Val)
{
 for(int i=0;i<_numOfElements;i++)
 {
    if(_arr[i] == Val)
    return i;
 }
 return -1;
}

//Array indexing operator
int& IntArray::operator[](int index)
{
    if((index < 0) && ( index > _numOfElements - 1))
    return _arr[_size];
```

```
    else
    return _arr[index];
}


//This is overloaded + sign that adds two arrays and return the result
IntArray operator+(IntArray& Array1,IntArray& Array2)
{
 int A1Count=0,A2Count=0,A3Count=0;
 IntArray Array3(Array1._size);

 // Copy the elements of the first array into the temporary array.
 while(A1Count <  Array1._numOfElements)
 Array3.add(Array1._arr[A1Count++]);

 // Add the contents of the second array to the temporary array.
 while(A2Count < Array2._numOfElements)
 Array3.add(Array2._arr[A2Count++]);

 return Array3; // Return the resultant array.
}

// This function checks to see if the given value is present in the array.
// If it is present, 1 is returned , if it is present 0 is returned
int IntArray::notContains(int Val)
{
 for(int i=0;i<_numOfElements;i++)
   if(_arr[i]==Val) return 0;

       return 1;
}

// This is an overloaded subtraction operator that takes two IntArrays as
// arguments and returns the result of subtracting the second from the first
IntArray operator-(IntArray Array1, IntArray Array2)
{
   IntArray temp;
   temp = Array1;

   for(int i=0;i<Array2._numOfElements;i++)
    temp.rmValue(Array2[i]);

   return temp;
}

//Overloaded & operator. This function takes two IntArrays
// as argument and returns a 1 if they have one common value.
int operator&(const IntArray& Arr1, const IntArray& Arr2)
{
 for(int i=0;i<Arr1._numOfElements;i++)
 {
    for(int j=0;j<Arr2._numOfElements;j++)
    {
     if(Arr1._arr[i] ==Arr2._arr[j])
     return 1;
    }
 }
 return 0;
}

// Overloaded = operator
IntArray& IntArray::operator=(const IntArray& Arr)
{
   if(_arr != NULL) // If memory was allocated before and not freed,
   {                // free the memory.
    delete [] _arr;
    _arr = NULL;
   }

   _size =Arr._size;
   _numOfElements = Arr._numOfElements;

   _arr = new int[_size];
   assert(_arr);

  for(int i=0;i<_numOfElements;i++) // Copy the elements.
```

```
  _arr[i]=Arr._arr[i];

  for(i= _numOfElements; i<_size;i++) // In the blank locations put -1. This is
  _arr[i]=-1;                        // to make debugging easy. In case an unneeded array
                                     // location is accessed.
  return *this;
}

IntArray::~IntArray()
{
   delete [] _arr;
   _arr = NULL;
   _size = 0;
   _numOfElements=0;
}
```

VITA

Kamalakar Ananthaneni

Candidate for the Degree of

Master of science

Thesis: GENERATION OF MAXIMALLY PARALLEL TASK SYSTEMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Vijayawada, India, July 28, 1973, son of Drs. Narayana Rao and Bharathi Ananthaneni.

Education: Received Bachelor of Engineering in Computer Science from Kuvempu University, Karnataka, India, in September 1994. Completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department at Oklahoma State University in December 1997.

Professional Experience: Graduate Assistant for the Office of International Students and Scholars, Oklahoma State University, July 1996 to October 1997.