NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle
A Visual Database System for Image Analysis on Parallel Computers and its Application to the EOS Amazon Project

5. Report Date

6. Performing Organization Code

7. Author(s)
Linda Shapiro

8. Performing Organization Report No.

10. Work Unit No.

9. Performing Organization Name and Address
University of Washington
3917 University Way NE
Seattle, WA 98105-6692

11. Contract or Grant No. NAS5-32337
USRA subcontract No. 5555-21

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001
NASA Goddard Space Flight Center
Greenbelt, MD 20771

13. Type of Report and Period Covered Final
August 1993 - July 1996

14. Sponsoring Agency Code

15. Supplementary Notes
This work was performed under a subcontract issued by
Universities Space Research Association
10227 Wincopin Circle, Suite 212
Columbia, MD 21044 Task 21

16. Abstract

The goal of this task was to create a design and prototype implementation of a database environment that is particular suited for handling the image, vision and scientific data assoicated with the NASA's EOC Amazon project. The focus was on a data model and query facilities that are designed to execute efficiently on parallel computers. A key feature of the enviroment is an interface which allows a scientist to specify high-level directives about how query execution should occur.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
database enviroment for EOS Amazon project	Unclassified--Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	1	

NASA Form 1626 Oct 86

A Visual Database System for Image Analysis on Parallel Computers and its Application to the EOS Amazon Project Subcontract No. 555-21

Linda G. Shapiro, Steven L. Tanimoto, and James P. Ahrens

Department of Computer Science and Engineering

University of Washington

Box 352350, Seattle, WA 98195-2350

Final Report
1994 – 1996
October, 1996

1 Introduction

1.1 Task Objective

The goal of this work was to create a design and prototype implementation of a database environment that is particular suited for handling the image, vision and scientific data associated with the NASA's EOS Amazon project. We are focusing on a data model and query facilities that are designed to execute efficiently on parallel computers. A key feature of the environment is an interface which allows a scientist to specify high-level directives about how query execution should occur. Using the interface does not require an understanding of the intricate details of parallel scheduling.

1.2 Introduction

This report summarizes research activities to date and serves as the final 3-year subcontract report. In the first year, we interviewed NASA scientists in order to understand their requirements and formulated an initial design for the database environment. In the second year, we refined the design and implemented a prototype. In the third year, we evaluated and documented the environment.

Our work was done in conjunction with the NASA Earth Observing System (EOS) Amazon Project at the University of Washington. The mission of the EOS Amazon project is to contribute to understanding the dynamics of the Amazon system in a natural state, and how it would evolve under possible change scenarios (from instantaneous deforestation to more subtle longer term climatic/chemical changes). The overall goal of the project is to determine how extensive land-use changes in the Amazon would modify the routing of water and its chemical load from precipitation, through the drainage system, and back to the atmosphere and ocean. The work is being undertaken by a number of groups here at the University of Washington including researchers in Hydrology headed by Thomas Dunne, in Biogeochemistry headed by Jeffrey Richey and Remote Sensing headed by John Adams.

1.3 Scientists' Requirements

We interviewed the NASA scientists in order to understand their computing requirements. The scientists are working with data sizes on the order of hundreds of megabytes and processing algorithms whose completion time is on the order of minutes to hours. The scientists identified the following desirable properties for a computing environment to support scientific

research:

- **Exploratory** – The computing environment should facilitate the scientist's exploration of different algorithmic solutions.
- **Responsive** – Algorithm results should be returned as quickly as possible especially if the scientist is waiting for them.
- **Satisfies user requirements** – The environment should schedule and execute algorithms based on the scientist's requirements for resource utilization and algorithm execution. For example, a scientist might like to specify which results are most important, what processing resources are available and how to utilize these resources.
- **High-level** – The environment's interface should let the scientist specify a high-level description of his algorithms and requirements. The environment should provide support for scientists who are not computer experts.
- **Organized** – The computing environment should record and organize the scientist's computer-based research work for later retrieval.

1.4 Approach

The scientific computing environment described in this report has these desirable properties. The approach we used to create this environment contains the following steps:

1. An identification of how existing software tools fulfill the requirements described above.
2. Creation of new algorithms and tools which fill the gap left by existing software tools.
3. Integration of all these tools into a seamless whole.

In summary, we have identified two keys areas which are not well supported by existing software. These areas are:

1. **Support for automated parallel program scheduling and execution.**

   To achieve high-performance, programs are scheduled and executed on multiple processors. Parallel scheduling is a complex problem and automation is a welcome solution for scientists. One disadvantage of traditional tools is that they optimize for a fixed collection of preset scheduling goals. Another is that they do not fully automate the

scheduling process. An automated scheduling system which is responsive to the scientists' scheduling needs would improve both scientists' satisfaction with their computer systems and their productivity.

2. **Support for scientific experimentation.**

   An environment needs to provide a computer-based framework for scientists' interactions. One typical interaction that scientists perform is parameterized experimentation with their programs. This experimentation helps the scientist to understand the effects of input parameter and coding changes. With automated support scientists could focus on analyzing their experimental results instead of the process required to generate the results.

1.5 Background

This section presents a high-level summary of existing software tools including programming languages, systems and databases, which scientists use to support their computer-based research work. This overview details how existing software tools fulfill the scientists' requirements and where they fall short. In addition, it provides a context for understanding how the computing environment described in this report builds upon and relates to existing tools.

1.5.1 Languages

Scientists have traditionally used sequential, imperative programming languages such as FORTRAN to express their scientific algorithms. Although FORTRAN is a low-level language it is the language of choice for most scientists. One reason for this is that it is fairly straightforward to express efficient programs based on arithmetic expressions. It is one of the few programming languages which provides standardized support for complex arithmetic. Another reason is there is a legacy of FORTRAN programs that has been developed by scientists over the years. Scientists are very interested in reusing these programs, leveraging their work upon these existing successful programs.

An important advance in programming languages for scientists is visual programming languages. One of the most successful type of visual languages are data-flow-based visual programming languages. Examples include languages such as AVS [28] and Cantata/KHOROS [23]. Programs are expressed graphically as data-flow-based program graphs. Users can manipulate the program graph interactively, by adding and deleting tasks. Users have access to a library of existing tasks which are ready for use in their programs. These languages simplify program creation and the reuse of existing tasks. They support exploratory programming because changes can easily be made to programs without re-compiling.

One useful addition to a visual programming environment is support for parallel program scheduling and execution. Researchers at the Boeing Company created a data-flow based visual programming environment called Access Manager which allows distributed task execution [24, 9]. The first version of Cantata/KHOROS (version 1.0) also allows users to execute different tasks of their programs on different processors. Users are require to specify the details of this assignment. CM/AVS is an extension of AVS in which a parallel version of program tasks can be executed on the Thinking Machine CM-2 or CM-5 parallel computers[2]. Support for parallel program execution is a necessary first step in the process of providing support for automated parallel scheduling and execution.

1.5.2 Systems

Another way a scientist can improve his program's efficiency is to use distributed system software tools, such as Condor [19] or DQS. These tools execute a set of independent jobs on networks of workstations. The scientist formulates his program as a collection of independent jobs and submits them to a job queue. The tool then automatically schedules and executes the jobs on a set of available workstations. Work continues on the creation of efficient distributed systems support tools. Recent research focuses on methods of identifying and using idle workstations and avoiding scheduling conflicts[5].

There are many task scheduling algorithms that can be used to schedule the tasks of a data-flow program graph in parallel. Task scheduling algorithms attempt to maximize the number of tasks executing in parallel while minimizing inter-processor communication costs. A taxonomy of task scheduling algorithms can be found in [8]. Lewis et al [12] also provides a useful introduction to task scheduling. Since most types of task scheduling problems are NP-complete, solution algorithms are based on heuristics. Traditionally these heuristics optimize for a fixed preset collection of goals. This is a problem if the scheduling goals of the algorithms conflict with the scheduling goals of the user.

1.5.3 Databases

Databases provide support for storing, organizing and accessing scientific data. Key features of a database are its data model, which describes the stored data's relationships and semantics, and its query model, which describes how to retrieve the stored data. The relational data model represents data by tables of attributes. It is a simple model and popular for representing business data. Scientific data usually has more complex relationships than can be expressed using the relational model. Another concern is that scientific data, such as images and multi-level data structures, do not map well to relational tables. A second popular data model is the object-oriented model. Data is represented as a collection of data-structure-

based objects. The object-oriented model usually lacks effective query models, because the structures it represents are so diverse that it is difficult to query them efficiently.

Recent research has focused on creating a data model and database system which supports scientists' needs. Examples include GAEA [15], MDBS [27] and DEVR [26]. These systems combine features of the relational and object-oriented data model, striving for the simplicity of the relational model with the expressiveness of the object-oriented model.

1.5.4 Constraints

A constraint expresses a relationship the user would like to hold in the solution of a particular problem. The environment described in this report uses constraints to express the user's task scheduling preferences. Related environments that use constraints include geometric layout systems[22, 6], user interface builders[21] and machine vision systems [25].

An active area of research in constraint satisfaction is how to solve over-constrained systems (i.e. a set of constraints for which there is no solution that satisfies all constraints)[16]. Freuder and Wallace [14] adapt standard backtracking and consistency checking algorithms to satisfy a maximal subset of the constraints. Borning *et al* describe another solution to this problem: the user arranges his constraints in a hierarchy[7]. In the event that all constraints cannot be satisfied, the constraints at a higher level in the hierarchy are satisfied before constraints at a lower level in the hierarchy. The constraints at the top level are called requirements (or hard constraints) and must always hold. The constraints at the lower levels are called preferences (or soft constraints) and are satisfied based on their level in the hierarchy. Constraints within a level are solved based on a relative weighting provided by the user. The user-directed scheduler described in this report fits into this paradigm. The scheduler has two levels: the requirement level and one preference level. Preferences are satisfied based on their relative weights. Future work could consist of allowing the user to express a hierarchy of constraints to the scheduler, so that the user can control the order of constraint satisfaction.

1.5.5 Artificial-Intelligence-based Scheduling

Scheduling is the process of assigning a set of jobs to set of limited resources over time. The quality of a schedule is usually defined by a collection of user-defined criteria and constraints. Artificial Intelligence (AI) is the study and creation of theory, algorithms and computer systems that use knowledge and encoded intelligence to solve complex problems. Thus, scheduling is a natural area of interest for researchers in Artificial Intelligence.

AI researchers have built scheduling systems for a number of specific domains including

systems for scheduling telescope usage [17], space shuttle maintenance [30], manufacturing [13] and defense logistics [10]. AI-scheduling solution methods are characterized by a number of features. *Constructive methods* build a complete schedule while *repair-based methods* incrementally update an existing but flawed schedule until a valid schedule is obtained. Fox's ISIS manufacturing scheduling system use a constructive solution method [13]. It iteratively builds a complete schedule by exploring a search space of partial schedules. It uses a beam-search which is guided by system and user constraints in order to find a schedule. Repair-based methods are useful for domains which change significantly over time. Repair-based methods only need to reschedule tasks affected by an external change to the problem. Zweben *et al* describe a repair-based scheduling system for space shuttle repair and maintenance[30]. It also uses a search-based solution method but explores a search space of complete schedules. A disadvantage of repair-based methods is that they usually use a local search-based solution method and therefore do not provide globally optimal schedules.

Many AI schedulers use constraints to express requirements and preferences on the problem domain. A characteristic of a scheduler is how it relaxes the problem constraints when they are in conflict in order to find a solution. Different methods include satisfying a maximal subset of constraints[17], using a fallback constraint if the original constraint cannot be satisfied [13], placing priorities on constraints and using a hierarchy of constraints[7].

The goal of this work is to create an automated task-scheduling environment. A critical component of the environment is a unique AI task-scheduler which allows the user to express task-scheduling constraints. The task-scheduling domain is different than other studied AI scheduling domains. For example, there are significant differences between the tasks in the task-scheduling domain and the jobs in the manufacturing domain. Tasks in a task-scheduling domain can usually be assigned to any processor whereas jobs in the manufacturing domain are assigned to specific machines. In the task-scheduling domain, if dependent tasks are scheduled on different processors a communication cost is incurred. There is no similiar cost in the manufacturing domain. Furthermore, tasks in the task-scheduling domain usually do not have start and finish deadlines as jobs do in the manufacturing domain. Because of the many required manufacturing constraints, problems in the manufacturing domain are usually over-constrained. Therefore solution methods usually focus on finding an acceptable solution. Problems in the task-scheduling domain are usually significantly less constrained and therefore this work uses a constructive solution method which can often provide an optimized solution to its users.

1.5.6 A related environment

The members of the Intelligent Data Management Project led by Nicholas Short at NASA Goddard are working on a prototype environment which can process the massive datasets generated by satellites that are part of NASA's Earth Observing System [18]. The envi-

ronment supports the querying, real-time processing and storing of satellite image data. In order to cope with the changing volume of incoming satellite image data by a given deadline, the environment has access to different versions of processing algorithms, which offer varying tradeoffs of result quality for shorter completion times.

The major subsystems of the environment are a set of processing request queues, a planning system, an execution engine/monitor and an object database.

- The processing request queues accept processing requests from users. Their requests are high-level and declarative allowing a user to express what processing should be done rather than how. For example, a user can specify that a satellite image be registered without specifying a specific algorithm to do the registration. A user can also specify a completion deadline for a processing request.
- The planning system inputs a processing request and selects and composes a set of tasks into a program graph which fulfills the user's request. The tasks are selected from a collection of Khoros tasks, LAS tasks [1] and user-defined tasks. Note that there are many tasks available to the planner, that perform the same type of operation but have different properties. For example, there may be multiple registration tasks: one which processes a specific image type, one which executes very quickly and one which produces registrations of very high quality. Task properties are formalized using conditions. Each task is annotated with a set of preconditions which must be true in order to execute the task and postconditions which are true after the task has executed. The planner unifies these conditions to create a program[3].
- The execution engine executes the program generated by the planner on a network of workstations. It uses a dynamic scheduling technique developed by Ma *et al* [20] to schedule based on network traffic, processor utilization and task dependencies.

In summary, Short *et al*'s environment supports automatic program creation by allowing users to express requests for processing which are fulfilled by a planner. Scheduling requests are limited to completion deadlines. In contrast, our own work allows its users to express a full range of task-scheduling directives including the ordering of program results and the specification of task assignments and processor utilization levels. In addition, this work supports computer-based scientific experimentation.

1.6 Structure of the Environment

Having reviewed existing scientific software tools, we will now describe the components of the scientific computing environment presented in this report:

[1] LAS is a geographic information system package used to process Landsat images.

- **Data-flow based visual programming environment** – The scientist uses a visual programming environment to construct his programs.
- **Scientific database** – A database is used to organize and store information about program graphs and results.
- **Distributed executor** – The executor executes a program graph in parallel on a network of workstations in order to quickly generate the scientist's results. It handles inter-processor communication between distributed tasks in the program graph and records performance information for use by the performance prediction tool.
- **Scheduler** – The scheduler automatically schedules a program graph on a network of workstations based on the scientist's directives. The scientist's directives are specified declaratively as constraints.
- **Performance prediction** – Program performance prediction is necessary for efficient scheduling. The scheduling algorithm uses performance estimates to make scheduling decisions.

A diagram of the scientific computing environment is shown in Figure 1. The diagram shows the data-flow between the components of the environment. In this report, data-flow diagrams are represented visually with boxes representing operations and ovals representing data. Directed arrows define the flow of data through the data-flow diagram.

Data input to the environment includes resource information, a program graph and the user's scheduling directives. Available processors are specified initially by the system administrator. The program graph is specified using a visual programming environment. The user scheduling directives are specified using a constraint-based scheduling language. The program graph and resources are used by the automatic performance prediction tool to create a cost model of program execution and processor utilization. The scheduler inputs the resource information, the program graph, the user's scheduling directives and performance estimate information. The scheduler outputs a schedule which fulfills the user's scheduling directives. The program is then executed on a network of workstations using the distributed executor. During execution, performance data is collected and sent to the performance database for future use by the performance prediction tool.

1.7 Outline

Section 2 describes a problem space representation for task scheduling. This goal-oriented representation facilitates the specification of scheduling directives. It contains the definition of a language for specifying these directives and a number of examples, which show how to

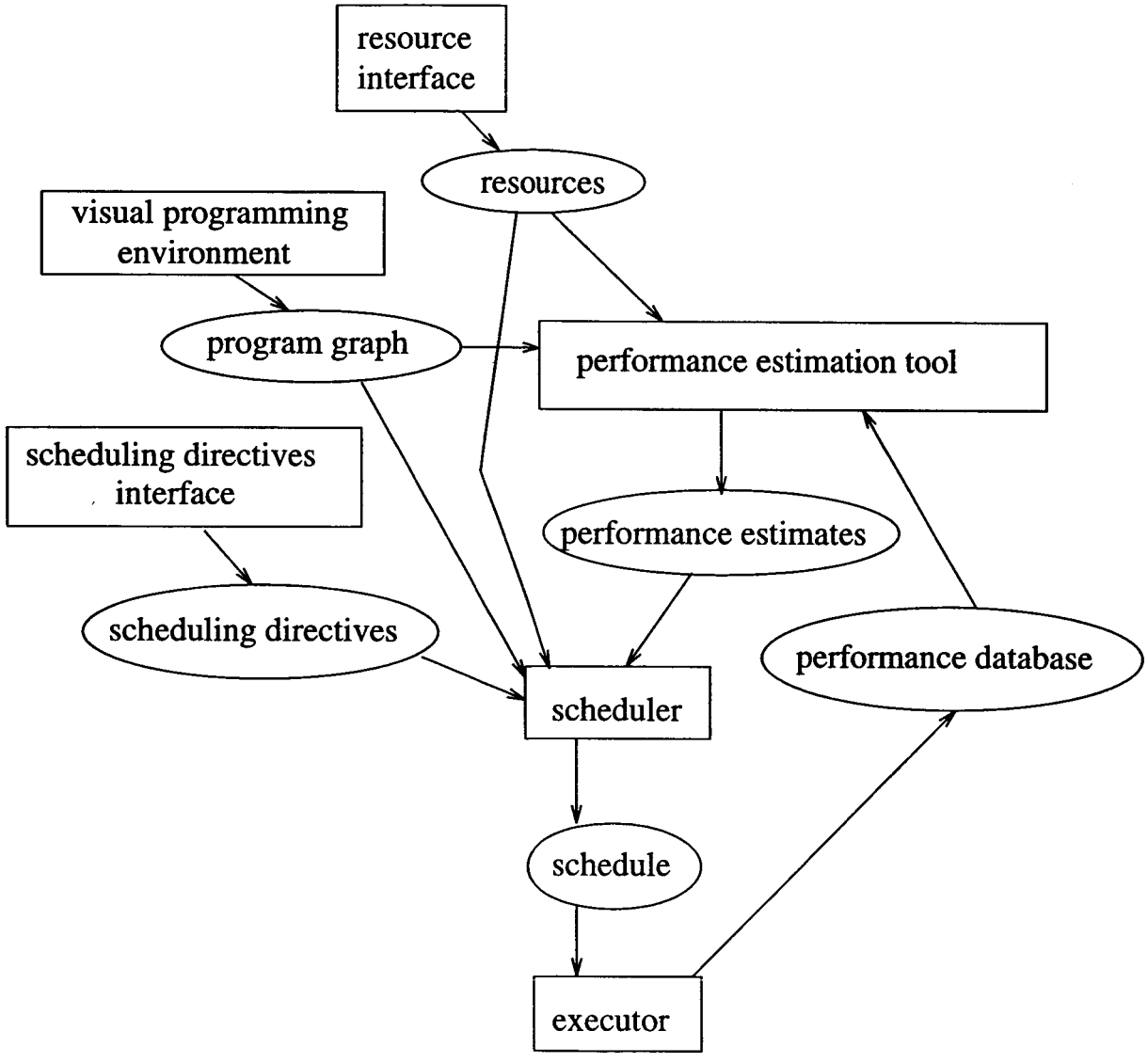

Figure 1: Structure of the scientific computing environment

use the language to specify directives for task ordering, task placement, processor utilization and load balancing. In addition, it describes a search-based algorithm for fulfilling a user's scheduling directives. Section 3 describes the prototype and an algorithm for automatically creating parameterized scientific experiments. Section 4 reports the results of a study of the environment performance. Results are presented on the performance of the environment on a large number of realistic imaging graphs and on how well the environment fulfills the user's scheduling directives. Section 5 summarizes and describes future research directions.

2 User-directed scheduling

To achieve high performance, programs are scheduled and executed on multiple processors. Parallel scheduling is a complex problem and automation is a welcome solution for scientists. One disadvantage of traditional tools is that they optimize for a fixed set of preset scheduling goals such as simply minimizing completion time. Another is that they do not fully automate the scheduling process. A method for automatic scheduling which is responsive to their scheduling needs would improve both scientists' satisfaction with computer systems and their productivity.

This chapter describes an automatic scheduling method that was designed to meet these needs. First, a problem space representation for scheduling is described. This goal-oriented representation facilitates the specification of scheduling directives and is amenable to artificial-intelligence-based solution techniques including search and planning. Then a language for specifying scheduling directives is defined. Finally, a search-based algorithm for determining a schedule is described.

2.1 Preliminaries

A program graph consists of a set of functional tasks and set of input and output dependencies between these tasks. Figure 2 shows an example of a simple program graph with two tasks, one which inputs an image and another which displays an image. The output of the Input image task is used as input by the Display image task.

Task scheduling is the process of assigning and ordering the execution of tasks from a program graph onto a collection of processors. The parallel task execution model used by the environment assumes that each processor can run one task at a time. To execute a task on a processor:

1. All inputs that are the outputs of tasks executed on another processor in the distributed

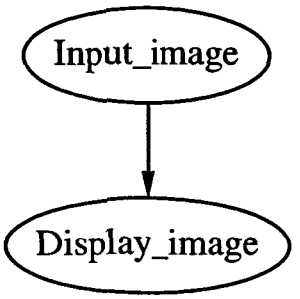

Figure 2: A program graph

network are received in parallel. The processor blocks and waits until all inputs are received.

2. The task is executed.

3. All outputs that are inputs of a task executed on other processors in the distributed network are sent to these processors in parallel.

Blocking communications assure the correct parallel execution of the task graph by guaranteeing a task is not executed until all its inputs are available.

2.2 A Problem Space Representation for Task Scheduling

This section describes a problem space representation for task scheduling. A problem space is defined as a set of states and operators that moves between these states. A particular problem to be solved in a problem space is known as a problem instance and is defined by an initial state and a set of goal states.

2.2.1 States

A state represents an empty, partial or complete schedule of tasks to processors. It must represent task and processor scheduling information as well as other related information such as estimates of scheduled task start and finish times. A state consists of a collection of tasks, a collection of task dependencies and a collection of processors. Elements of these collections are entities. Each entity consists of a set of attributes, each of which consists of a name and type. Attributes are detailed below using the following syntax: <attribute name>:<attribute type>; descriptive comment. The tasks, processors and task dependency entities are as follows:

Task Entity

```
id:integer                  ; a unique task id
name:string                 ; the task's name
exec-time:integer           ; the task's execution time
                            ; Note: all timings are expressed in seconds
start-time:integer          ; the task's start time
                            ; Note: the start of the schedule is time 0
finish-time:integer         ; the task's finish time
assigned-proc-id:integer    ; the id of the processor this
                            ; task is assigned to
```

Processor Entity

```
id:integer                  ; a unique processor id
name:string                 ; the processor's name
finish-time:integer         ; the total running time of
                            ; the tasks scheduled on this processor
                            ; assuming no gaps or idle periods
assigned-task-ids           ; an ordered list of tasks scheduled on
   :list                    ; this processor
util:integer                ; the processor's CPU utilization
```

Dependency Entity

```
task-id:integer             ; a task id
dep-task-id:integer         ; the id of the task that depends on
                            ; the output of the task with task-id
                            ; as input
comm-time:integer           ; the time to required to communicate this
                            ; dependency data
                            ; Note: if no communication is required than
                            ; comm-time is 0.
non-local-comm-time:integer ; the time to communicate this
                            ; dependency data to another processor in
                            ; in the network
```

2.2.2 Initial State

The initial state has the following values initialized:

- There is a task entity for each task in the input data-flow program graph.
- There is a processor entity for each identified available processor.
- There is a dependency entity for each dependency in the program graph.

All other attributes of these entities are assigned to a special symbol which represents unknown values.

2.2.3 Operators

An operator makes a transition from one state to another state. There is one operator in the problem space representation for task scheduling. Its name and type is: *schedule-task-to-processor(integer, integer, state)* $\rightarrow$ *state.* [2] The result of executing the call, *schedule-task-to-processor(task-id, proc-id,original-state)* $\rightarrow$ *new-state* is that the task identified by *task-id* is scheduled on the processor identified by *proc-id.*

2.3 Goal State

The conditions required of a goal state are:

- Each task is scheduled to a processor.
- The task dependencies are respected by the schedule. That is, if a task is dependent upon another task for input, it runs after that task has completed.

This completes the specification of a problem space representation for task scheduling.

[2] We will use the following syntax to describe function types in this document: <function name>(<param type 1>, <param type 2> etc.) $\rightarrow$ <return param type>.

2.4 A Language for Expressing Scheduling Directives

The problem space representation for task scheduling defines any complete valid schedule of tasks to processors as a goal state. Traditional task scheduling algorithms add another condition to these criteria. They optimize performance by working to minimize a particular performance variable, such as processor completion time, or task finish times. These optimizations are always hard-encoded into the scheduling algorithm, and these algorithms do not allow other optimization criteria to be used. In this section, I describe a language in which a user can specify a variety of optimization criteria, by describing relationships he would like to hold between values in the goal state and values he would like to be minimized or maximized in the goal state. These *scheduling directives* allow the user to optimize for performance as well as specify other desirable properties of a schedule including the ordering of task outputs, specific task to processor assignments and specific processor utilization levels.

2.4.1 Preliminaries

The scheduling language is an extension of SQL [1, 11] a relational database query language. SQL is the pre-eminent database language in use today, enjoying wide acceptance among non-computer experts because of its ease of use.

In SQL, a relation is a collection of entities with the same sets of attributes. A state in the task scheduling problem space representation is composed of three relations: tasks (`task`), processors (`proc`) and dependencies (`dep`).

A basic SQL expression has three clauses: **`select`**, **`from`** and **`where`**. The **`from`** clause specifies the relations to be operated on. The **`where`** clause specifies a boolean predicate on entity attributes which are used to select entities from the relations. The **`select`** clause specifies the resulting relation in terms of the attributes of the selected entities. The syntax is:

```
select <attributes from the selected entities>
from <relations>
where <boolean predicate on the entity attributes of the relations>
```

The scheduling language defines importance and type constraints. Importance constraints are either requirements or preferences. Requirements must always hold, preferences are fulfilled based upon user-defined priorities. Constraint types include relationship-based constraints that express a desired relationship between attributes of relations, value-based constraints that express a desire for a value to be minimized or maximized, and ordering-based

constraints that express a desire for a particular ordering on a relation. The basic syntax for constraints is:

```
assert {relationship | value | ordering} {requirement | preference}
(
<specific assertion constructs>
)
```

The bracket and slash notation used above (i.e {A|B|C}) means that one of the elements in the collection of choices is utilized. For example, valid constraints include: `assert value requirement` and `assert ordering preference`.

Selecting elements from a collection An SQL expression can be used to select entities which pass a given test. Using the * symbol in the select clause returns all the attributes of an entity. Note that, the attributes of a relation are referred to by appending the attribute name to the entity type name. For example, the *id* attribute of the *task* entity is *task-id.*

Aggregating the elements of a relation SQL also provides a way to compute a single summary value from a collection of attribute values. In the select clause the user identifies a specific attribute to aggregate. Possible aggregate functions include: average, minimum, maximum, sum and count.

2.4.2 Requirements

The first type of scheduling directive is a requirement. A requirement guarantees that a user-specified constraint will hold in a goal state. Requirements are specified and tested with a requirement function.

Relationship requirements A relationship requirement guarantees that a user-specified relationship will hold in a given state. The name and type of the relationship requirement function is:

assert relationship requirement (expression, test, expression) $\rightarrow$ *boolean.*

It returns TRUE when applied to a valid state. For the call, *assert relationship requirement(expression-1, test-1, expression-2)*:

- expression-1 and expression-2 are SQL expressions. The function applies the SQL expressions to the given state. The returned values are used to create *relation-1* and *relation-2.*
- *test-1* is run on each element of the cross product of the previously created resulting relations (i.e. all possible pairs of an input value from the first relation and an input value from the second relation). If any test returns FALSE the requirement is FALSE.

Example 1 - Ordering task output generation time To assert that the task with id 1 finishes before the task with id 2 the following requirement is defined:

```
assert relationship requirement (
(select task-finish-time from task where task-id = 1) <
(select task-finish-time from task where task-id = 2) )
```

Example 2 - Deadlines on task output generation time To assert that all tasks finish before a 30 seconds deadline the following requirement is defined:

```
assert relationship requirement (
(select task-finish-time from task) < 30 )
```

Example 3 - Controlling task/processor assignments To assert all FFT tasks are run on lillith the following requirement is defined:

```
assert relationship requirement (
(select task-assigned-proc-id from task where task-name = ''FFT'') =
(select proc-id from proc where proc-name = ''lillith''))
```

Ordering requirements An ordering requirement function provides a means for asserting relationships which hold on an ordered sequence of values. Thus, the relationship test holds between each element of the sequence and any subsequent elements. Its name and type are:

assert ordering requirement (sequence, ordering-test) $\rightarrow$ *boolean.*

For the call, *assert ordering requirement(sequence-1, ordering-test-1)*:

- The *ordering-test-1* is applied to *sequence-1*. The `order-test` clause is an extension to standard SQL, allowing the user to specify a sort order to test. The `order-test` clause identifies the attributes to test and whether to test if the sequence is sorted in ascending or descending order. If any entity of the sequence is out of order the ordering test returns FALSE.

Example - Ordering task output generation time To force the tasks to be scheduled in order of id number the following requirements is made: [3] [4]

```
assert ordering requirement (
(select * from task where task-assigned-proc-id <> UNKNOWN)
(order-test task-id asc))
```

Ordering-based requirement functions are useful for scheduling tasks to processors in a particular order. Many traditional task algorithms define an order in which to schedule tasks. With ordering-based requirement functions this behavior can easily be mimicked.

Additional goal state condition Requirements add an additional condition to the problem state representation of a goal state: when applied to a goal state all defined relationship and ordering-based requirements must be TRUE.

2.4.3 Preferences

Relationship and Ordering Preferences The second type of scheduling directive is a preference. A preference specifies a relationship the user would like to hold in a goal state or a value the user would like to minimize or maximize in the goal state. There are relationship and ordering based preference functions and they are very similiar to relation and ordering requirement functions. The only difference between these types of preference and requirement functions is their return values. Requirement functions return TRUE if all tests are passed and FALSE otherwise. Preference functions return the number of failed tests. The name and type of the relationship and ordering preference functions are:

assert relationship preference(expression, test, expression) $\rightarrow$ *integer* and

[3] Note that the order-by clause creates a sequence from the unordered relation using one key and the order-test clause tests if the sequence is ordered based on a different key.

[4] The order-by clause considers entities out of order if the *task-assigned-proc-id* value of the task earlier in the sequence is UNKNOWN and the *task-assigned-proc-id* value of the task later in the sequence is known.

assert preference order(expression, ordering-test) $\rightarrow$ *integer.* [5]

The ordering preference function computes for each element in the sequence the number of subsequent elements that should precede it in the specified ordering. The sum of these values is returned by the function. This calculation places decreasing emphasis on the correct ordering of entities as their distance from the beginning of the sequence increases.

Example 1 - Balancing the task load on processors To specify a preference for a balanced task load among the processors the following function is specified:

```
assert relationship preference
all (select task-start-time from task) <=
all (select proc-finish-time from proc)
```

This expression states that there is a preference that all task start times be less than the total running time of each processor. The intuition for why this balances workload is that in an unbalanced workload, tasks start on some processor after other processors have finished. Note that this relationship should not be expressed as a requirement because when communication costs are excessive, optimal schedules are not balanced.

Example 2 - Controlling processor utilization To specify a preference for the processor *calvin* to be assigned at least twice as much task load as the processor *lillith* the following function is specified:

```
assert relationship preference
2 * (select proc-finish-time from proc where proc-name = ''lillith'') <=
    (select proc-finish-time from proc where proc-name = ''calvin'')
```

Value-based preferences Value-based preferences allow the user to specify values they would like minimized or maximized in the goal state. The name and type of the value-based preference function is:

assert value preference(optimization-type, integer, function, integer, integer) $\rightarrow$ *integer.*

For the call *assert value preference(opt-type, priority, value-function, min, max)*:

[5] Requirements can be implemented with preferences as follows: `assert relationship requirement` calls `assert relationship preference` with the same parameters. If `assert relationship preference` returns 0 (tests failed) then return TRUE else return FALSE.

- *opt-type* states whether to minimize or maximize the value function.
- *priority* is a measure of the importance of fulfilling this preference. Specifically, priority values have the following semantics: The relative importance of a particular preference is equal to its priority value over the total of all priority values. For example, if three preferences have priorities, 1, 2, 1, the relative importance of the preferences is 0.25, 0.50, 0.25. For example, when choosing between two goal states, the environment will prefer a state which fulfills the second preference but not the first or third over a state which fulfills the first preference but not the second or third because the second preference is twice as important to the user as the first.
- *value-function* is a SQL expression which when applied to a given state returns an integer value.
- *min, max* are estimates of lower and upper bounds on the result of the *value-function.* These values are used by the environment to scale the result of the value-function so that comparisons with other value-function results make sense.

Example 1 - Minimizing processor run times To specify a preference for minimizing processor run times the following function is specified: [6]

```
assert value preference (
opt-type = minimize, priority = 1,
function = (select max(task-finish-time) from task)
min = 0, max = (select sum(task-exec-time) from task) +
   (select sum(non-local-comm-time) from dep))
```

All relationship and ordering-based preferences are expressed using value-based preferences because the environment can use value-based preferences to create a numeric measure of how much a state is preferred.

Additional goal state condition Preferences add an additional condition to the problem state representation of a goal state: goal states which fulfill preferences based on their priority values are preferred. A formal description of how this condition may be met is described in the next section.

[6]The maximum finish time value is bounded by the serial execution of all tasks plus the serial non-local communication of all dependency data.

2.5 A Search-based Scheduling Algorithm

In this section, I describe a search algorithm for user-directed scheduling. Best-first search is used to find optimized goal states in the problem-space representation. A best-first search algorithm requires three functions: a successor function, which defines how to create the successor states of a state, an evaluation function, which gives each state a score, and a goal function, which identifies goal states.

Best-first search selects from the set of states generated so far the state with the minimum score. It checks if the selected state is the goal state, if it is then the state is returned. Otherwise the successors of the selected state are created and evaluated and the process continues.

2.5.1 Successor Function

The name and type of the successor function is: *successor(state)* $\rightarrow$ *set of states.*

For the call *successor(state1)* the function creates:

- *set1* – a set of all tasks that could be executed. This set is composed of each non-scheduled task whose dependent tasks are already scheduled.
- *set2* – a set of all processors on which the tasks could be executed. This set is a list of all the available processors.

For all pairs of elements, *ele1* $\in$ *set1* and *ele2* $\in$ *set2*, *scheduled-task-to-processor(ele1, ele2, state1)* is executed. These executions create a set of new states.

All defined requirements are applied to each new state. If any requirement fails when applied to a new state, the state is removed from the set of new states. After this is complete, the remaining set of new states are returned as successors.

2.5.2 Evaluation Function

Semantics of priorities Preferences provide a mechanism for comparing states. For a call, *assert value preference(opt-type, priority, value-function, min, max)* the *opt-type*, *priority*, *min* and *max* values allows the environment to scale the results of value functions so that comparisons make sense. The following variables are used to calculate a global preference comparison value, g_{total} for a state from a set of $1 \ldots vp$ value-based preferences:

- p_i is the priority of preference i where $i = 1 \ldots vp$.
- p_{total} is the sum of all the preferences priority values, that is, $p_{total} = \sum_{i=1}^{vp} p_i$.
- v_i is the result of the value function of preference i.
- min_i, max_i is the lower and upper bound values of preference i.
- s_i is the scaled preference value of preference i (s_i values are between 0 and 1 with 0 preferred), that is, *if (type = minimize) then* $s_i = \frac{v_i - min_i}{max_i - min_i}$ *else* $s_i = \frac{max_i - v_i}{max_i - min_i}$.
- g_i is the scaled prioritized value of preference i, that is, $g_i = s_i * \frac{p_i}{p_{total}}$.
- g_{total} is the sum of all the preferences scaled prioritized values, that is, $g_{total} = \sum_{i=1}^{vp} g_i$.

The name and type of the evaluation function is: *evaluation(state)* $\rightarrow$ *integer.* The evaluation function returns the global preference value, g_{total} defined in the previous section. Best-first search find an optimized goal state but not necessary the optimal goal state because it stops as soon as it finds a goal state. Branch and bound search could be used to find the optimal goal state but the extra time it requires to search through the problem space is prohibitive.

2.5.3 Goal Function

The name and type of the goal function is: *goal(state)* $\rightarrow$ *boolean.* The goal function returns TRUE if all the tasks are scheduled and FALSE otherwise.

2.5.4 Soundness and Completeness

- Soundness is the property that if a goal state is returned by the search it is valid. Informally, this is true because:
  - Only valid states are identified as goal states since the goal function only returns TRUE if all tasks are scheduled.
  - Only valid states are generated because the successor function only schedules tasks whose dependent tasks have already been scheduled.
  - Only valid states are generated because the successor function eliminates states which do not satisfy the user's requirements.
- Completeness is the property that if a goal state exists it can be found by the search.
  - The successor function lists all valid task-to-processor assignments. Thus, all possible valid schedules can be generated.

2.5.5 Computational Complexity

The computational complexity of a search algorithm is the branching factor raised to the depth of the search tree (i.e. $O(b^n)$ where b is the branching factor and n is the depth). Let *tasks* be the number of tasks and *procs* be the number of processors. The worst case computational complexity is $O((tasks \times procs)^{tasks})$. The average computational complexity is usually better than this, because the branching factor is usually significantly less than the total number of tasks. The removal of states that do not meet the user's requirements further reduces the branching factor. A study of the performance of this algorithm on a large number of imaging graphs is presented in Section 4. The study reports on the number of states the algorithm generates.

3 SCE: The prototype

This section describes a prototype of the scientific computing environment SCE developed in this research. The first subsection describes how the prototype supports computer-based scientific experimentation. The second subsection describes how the user interacts with the prototype and the outputs that are generated. The last subsection presents an overview of the implementation of the prototype.

3.1 Computer-based Scientific Experimentation

Scientists are interested in experimenting with their programs. They make parameter and coding changes to their programs and then analyze their results in order to understand the effects of these changes. With automated support, scientists can focus more on analyzing their experimental results than on how to generate these results. This section describes how SCE supports computer-based scientific experimentation. An efficient algorithm for automatically creating a computer-based experiment is presented. This is followed by a discussion of another environment which provides support for experimentation and the specific advantages of the prototype's implementation.

An experiment specifies the controlled substitution of tasks, data or parameters in the program graph. [7] All possible combinations of substitutions may need to be tested. For example, a geologist working on a remote sensing problem might be interested in testing

[7] In most data-flow based visual programming environments, parameters and data are not represent explicitly in a program graph. Instead they are considered part of each task. For example, parameters and data in Cantata/Khoros are specified as input values. Thus, to specify parameter and data substitutions a corresponding task is specified with modified input values.

the quality of a set of edge detection tasks on a collection of satellite images. Using the prototype's visual programming environment, a program graph is created which consists of nodes for an input image task, edge detection task and display-image task connected as a sequence. The created program graph is shown in Figure 3.

In the experiment, the first task, Input_image_region_a, which contains data for the northern region of the Amazon river basin, is to be replaced with Input_image_region_b, which contains data for the southern region of the basin. The second task, the Sobel edge detector is to be replaced with two different edge detection tasks: the Prewitt edge detector and the Canny edge detector as shown in Figure 4. All possible combinations of substitutions of input images and edge detection tasks are instantiated and executed as shown in Figure 5. The output images are labeled and stored in the database for later analysis.

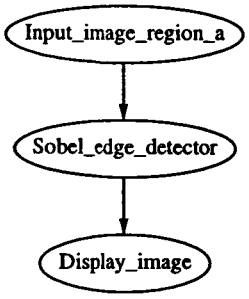

Figure 3: A sequence of tasks in a program graph

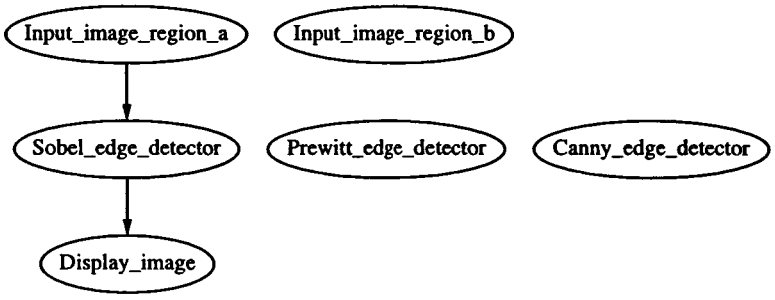

Figure 4: Substitutions for the experiment

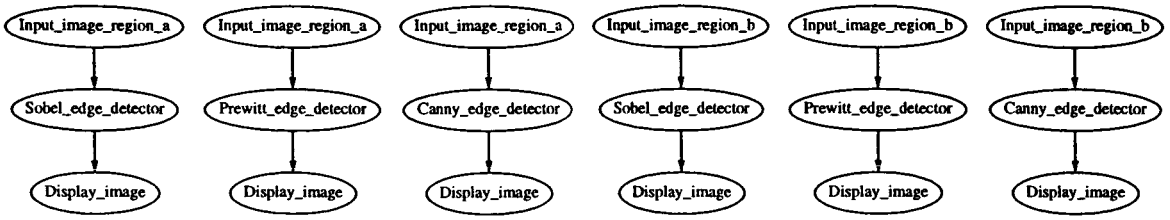

Figure 5: An instantiated experiment

A simple way to create an experiment is to replicate the original program graph for each possible combination of substitutions and then make one set of substitutions to each replicated graph. This method was used to create the experiment shown in Figure 3. This simple method requires more task executions than are necessary. For example, in Figure 3 notice that the Input_image_region_a task is executed three times although it is only necessary to execute it once. SCE uses a new experiment creation algorithm that avoids this problem by reusing the results of executed tasks. Reusing task results helps to minimize experiment execution time.

3.1.1 Discussion

A related environment which executes experiments on a collection of distributed workstations in parallel was created by D. Abramson *et al*[4]. The environment, Nimrod, allows a user to express a set of input parameters and data changes for a program. Nimrod creates experiments in a similiar manner to the example shown in Figure 3.3. The cross-product of user parameter changes is generated and elements from this set are input to copies of the original programs. These copies are scheduled and executed on a collection of distributed workstations.

Nimrod and SCE both provide a concise and useful interaction model. Experiments provide a concise method for scientists to express a set of controlled changes to a program graph. With this support, scientists can express what changes they want to experiment with, but not how to implement these changes. Nimrod and SCE also both provide efficient experiment executions. Experiments execute efficiently because their program graph representations contain many independent execution paths which can be scheduled and executed in parallel.

In addition, SCE simplifies experimentation with task substitutions in a program. Nimrod allows its users to experiment with different data and parameter inputs to their programs. Nimrod has no knowledge about the inner workings of the program on which it is running experiments. Thus, in order to make a task substitution in Nimrod, a scientist must modify his program by hand, removing the code to be substituted for, replacing it with new code and recompiling their program. After this process is complete he can use Nimrod to run experiments. In SCE, programs are represented as a collection of communicating tasks. SCE allows its users to experiment with program tasks. Thus, it is a simple matter to have the user identify which task to replace and to automatically substitute the user's new task in its place. Specifying task substitution is useful when the user wants to experiment with a collection of different tasks which perform the same function, such as edge detection.

Furthermore, SCE reduces the total amount of work required to execute an experiment. SCE uses an experiment creation algorithm which reuses task results whenever possible during an experiment. This algorithm allows scientists to obtain their experimental results faster than

Nimrod's experiment creation algorithm. Nimrod's algorithm replicates the entire program for each substitution. It cannot optimize the experiment creation process because it does not have any knowledge of the inner workings of the program on which it is running experiments.

3.2 A Sample Session with SCE

A sample user session with SCE is now presented. This includes a description of the components of SCE the user interacts with and the results of this interaction. This presentation helps the reader become familiar with the interface provided by SCE.

3.2.1 User Inputs

Visual program environment The scientist uses the visual programming environment, Cantata[23], to construct his programs. Figure 6 shows a Cantata workspace. The boxes represent tasks and lines connecting the tasks represent dependencies. The user selects tasks f rom the pull-down libraries at the top of the screen and connects the tasks together using dependencies to form a program graph.

Scheduling directives interface The scientist uses a text editor to express his scheduling directives. A set of default directives are supplied by the environment. Note that these directives do not have to be utilized, they are provided as a suggestion. The goal of these directives is to minimize program completion time. The default directives are described in more detail in Section 4.

Resource interface The scientist uses a text editor to create a list of available processors.

Experiment interface The scientist currently defines an experiment using a text editor to specify locations for task substitutions and sets of the tasks to substitute into the program graph. Future work on the environment could consist of modifying Cantata's interactive graphical interface to allow the user to express experiments graphically. Another useful feature would be to extend the experiment creation interface to allow the user to express experiments which do not create the full cross product of task substitutions. This is useful when the user is not interested in all experimental results. For example, in the example shown in Figures 3-5 the user may only be interested in testing the Sobel edge detector on Input_image_region_a and in testing the other edge detectors on both images.

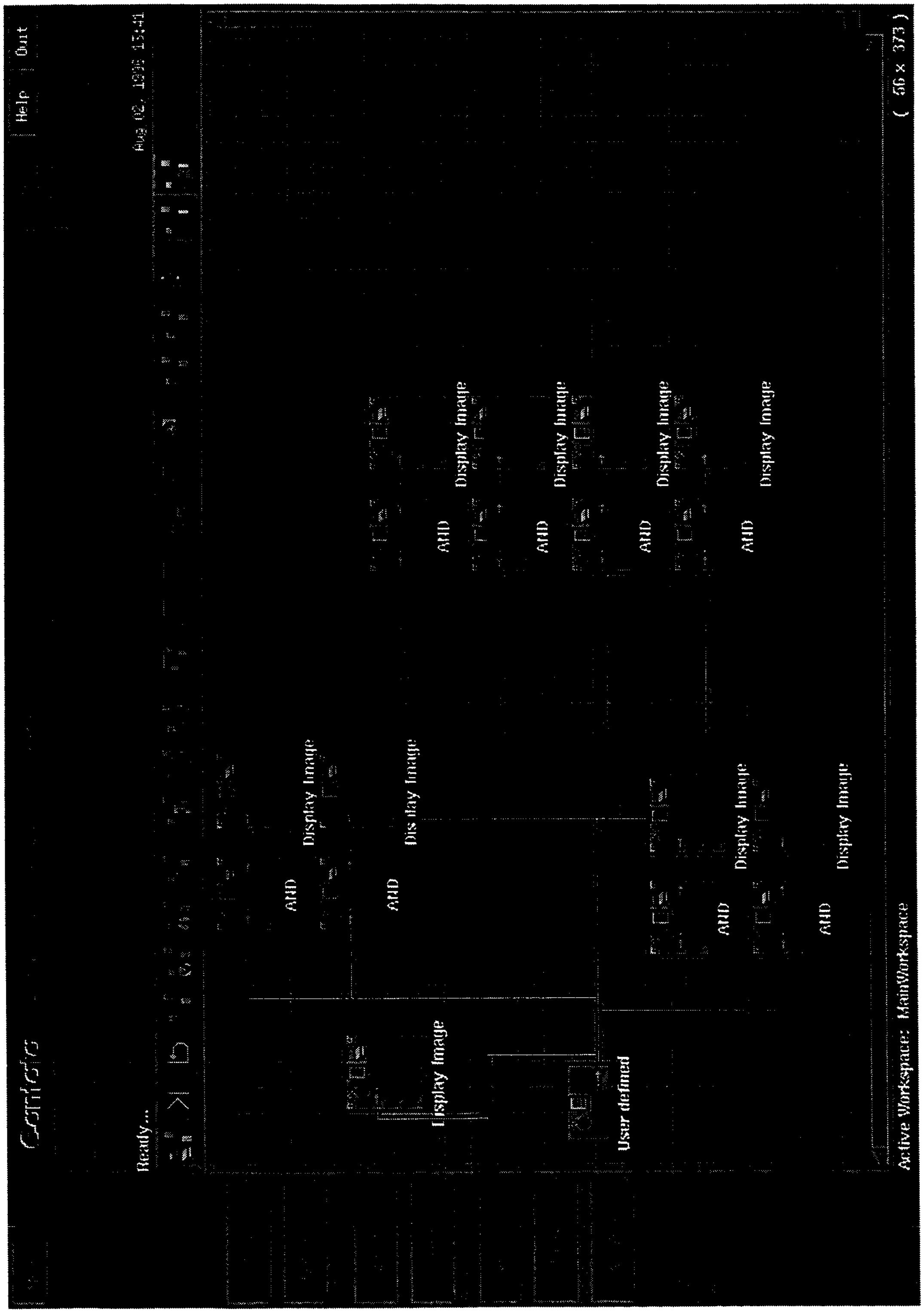

Figure 6: The Cantata visual programming environment

3.2.2 Prototype Results

After the user creates his program graph, scheduling directives and resources, SCE uses this data to create performance prediction information. This information is then passed to the scheduler which creates a schedule. Once the schedule is created, the tasks are executed on the workstations and the program outputs are generated. SCE creates an information log which records the details of each run. Details include the program graph and directives used, the generated performance prediction information and schedule and the program execution statistics. The log is written in HTML and the user can browse the information with a browser such as Netscape Navigator. Figure 7 shows an example of the information log. Scheduling information is stored graphically as part of the information log.

- **---Information log---**

Date: Thu_Jul_18_15:40:58_PDT_1996

Program graph: /projects/3D/ahrens/DIP/one-oper/bit-slice/bit-slices.wk

Performance prediction tool:
Performance prediction results

User directives:
Assertions
Preferences

Scheduler:
Scheduling results

Processors used:
*oddvar
*puyallup
*chelan
*manastash
*norge
*lutefisk

Executor:
Execution results

Figure 7: Information log

This completes Section 3. Section 4 presents results of a performance evaluation of the environment which include results on the efficiency of the experiment creation algorithm and the performance of the environment when executing experiments.

4 Results

This section reports the results of a performance evaluation of the environment and survey of usefulness of the environment to scientists to support their computer-based scientific research work. The performance evaluation consists of three different studies. The first study explores the performance of the environment using the default scheduling directives on a diverse collection of image processing program graphs. Results are presented on the performance of the prediction tool, the scheduler and the executor. The second study investigates how well the environment responds to the user's scheduling directives. The third study examines the performance of the environment on computer-based scientific experiments.

4.1 Testing Method

The environment was tested by scheduling and executing a collection of program graphs which are a part of the Digital Image Processing (DIP) course for the cantata/Khoros visual programming environment. The course presents lessons on topics in image processing and provides forty-seven example program graphs for students to modify and execute. Topics include image representation, image manipulation, linear and non-linear operators and pattern classification. [8] The average number of tasks in the program graph is 18 and the average number of dependencies is 18. This data shows that the program graphs have a significant number of tasks and dependencies. All tests were executed on a collection of nine ethernet-connected Sun SPARCstation-IPXs.

4.2 Performance Study 1 - Default Scheduling Directives

The first study explores the performance of the environment using the default scheduling directives. The goal of these directives is to minimize program completion time. The default directives and their purpose are now described.

The first default directive requires the scheduler to only use processors with utilizations of less than or equal to three percent. This allows a program graph to execute efficiently without interference from other user's programs. The directive works by requiring that all processors with utilizations greater than three percent have their task assignment list be empty (i.e. equal to UNKNOWN). The second default directive directs the scheduler to prefer states with more scheduled tasks. This directive allows the search algorithm to make efficient progress. The next three directives emulate Wu and Gajski's task scheduling

[8] The Digital Image Processing course can be found on the World Wide Web at http://www.eece.unm.edu/dipcourse/.

algorithm[29]. The goal of their algorithm is to minimize program completion time. The algorithm first determines an order in which to schedule the tasks. Then, as each task is scheduled, the algorithm chooses the processor that allows its earliest start time. The ordering is computed as follows: for each task, the length of the longest path between the task and any output task is calculated. The path length is the sum of the execution times and non-local communication times of the tasks and dependencies on the path. The tasks are arranged in non-increasing order based on their calculated path lengths.

Using these scheduling directives, the environment executes the DIP course program graphs. Results are presented on the performance of the scheduler, performance prediction tool, and executor.

Figure 8 shows the number of states explored by the scheduler for the program graphs. Notice that for most graphs the environment explores less than five hundred states. [9] Thus, the scheduler, when using the default directives, only needs to explore a small portion of the search space.

Figure 9 presents the speedup achieved by the environment using the default directives for the collection of program graphs. [10] It is important to study the speedup achieved by the environment to assess the performance of the default directives. During the scheduling process, the utilization assertion selects the number of processors that have a utilization of 3 percent or less. From this set of selected processors, the scheduler then schedules tasks on a subset of these processors. This subset is called the scheduled processors. When the number of scheduled processors is equal to the number of selected processors, it is possible that the scheduler could have used more processors to obtain better speedup. These instances are identified in Figure 9 by a dot in front of the program graph name.

The speedup data is grouped according to the number of scheduled processors (i.e. all program graphs scheduled on one processor, all program graphs scheduled on two processors, etc.). Within each group, the data is sorted from worst speedup to best speedup. The average speedup achieved was 1.4 on an average of 2.8 scheduled processors. [11] Note that the speedup the scheduler can obtain is limited by the existing data-flow parallelism in the program graphs. It is also important to note that this speedup was achieved without user intervention. The user provides a program graph to the environment, and it is automatically scheduled and executed.

[9] A worst case estimate on the average number of states in the search state is $18 * 9^{18} = 162^{18}$.

[10] Note that the input data used by the program graphs in this test was expanded to be 36 times larger (i.e. a factor of six expansion on the row and columns of the input images) in order to simulate the massive data sizes used by scientists such as geologists working on remote sensing problems.

[11] The geometric mean is used to average normalized values such as speedups.

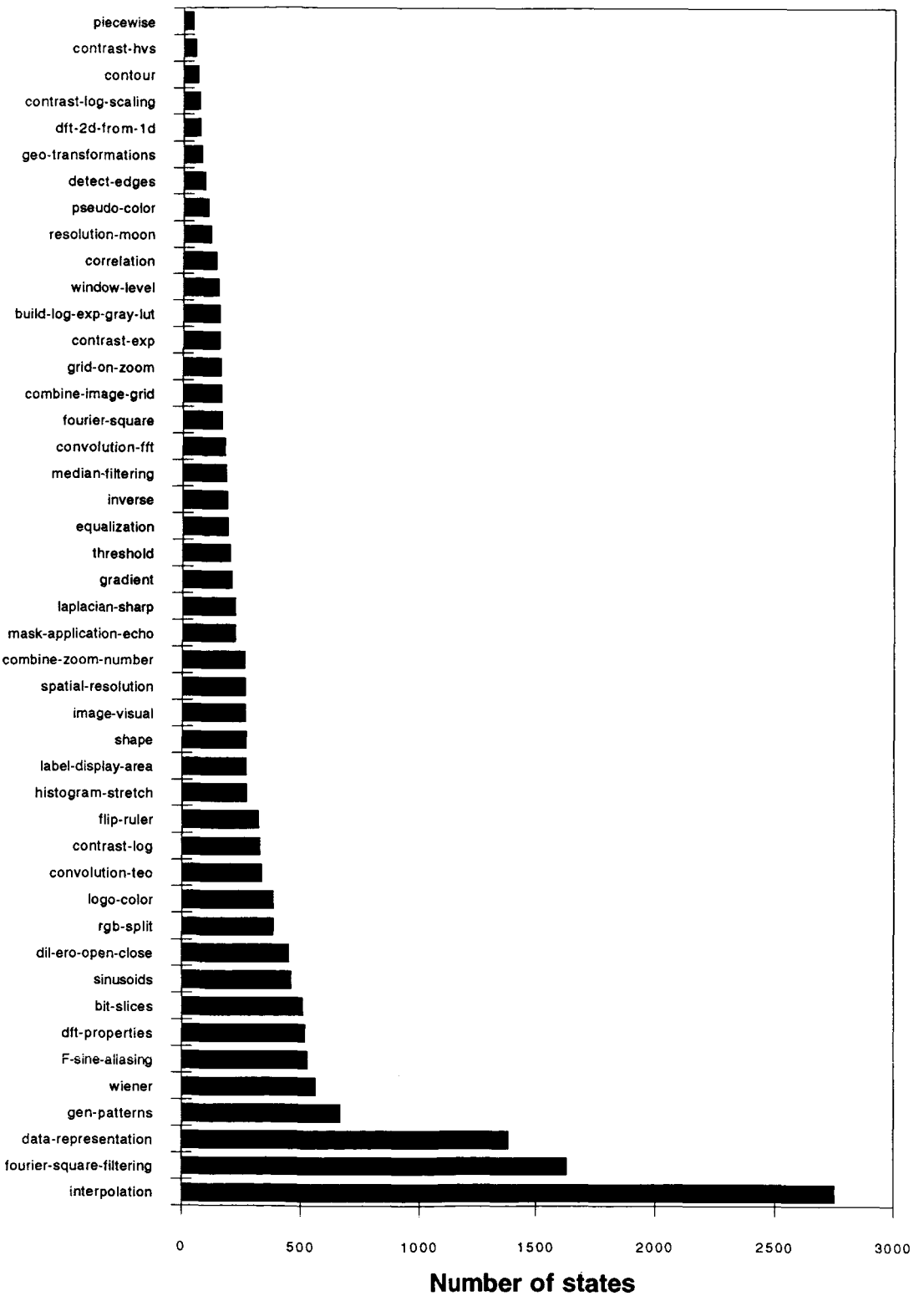

Figure 8: Number of states explored by the scheduler

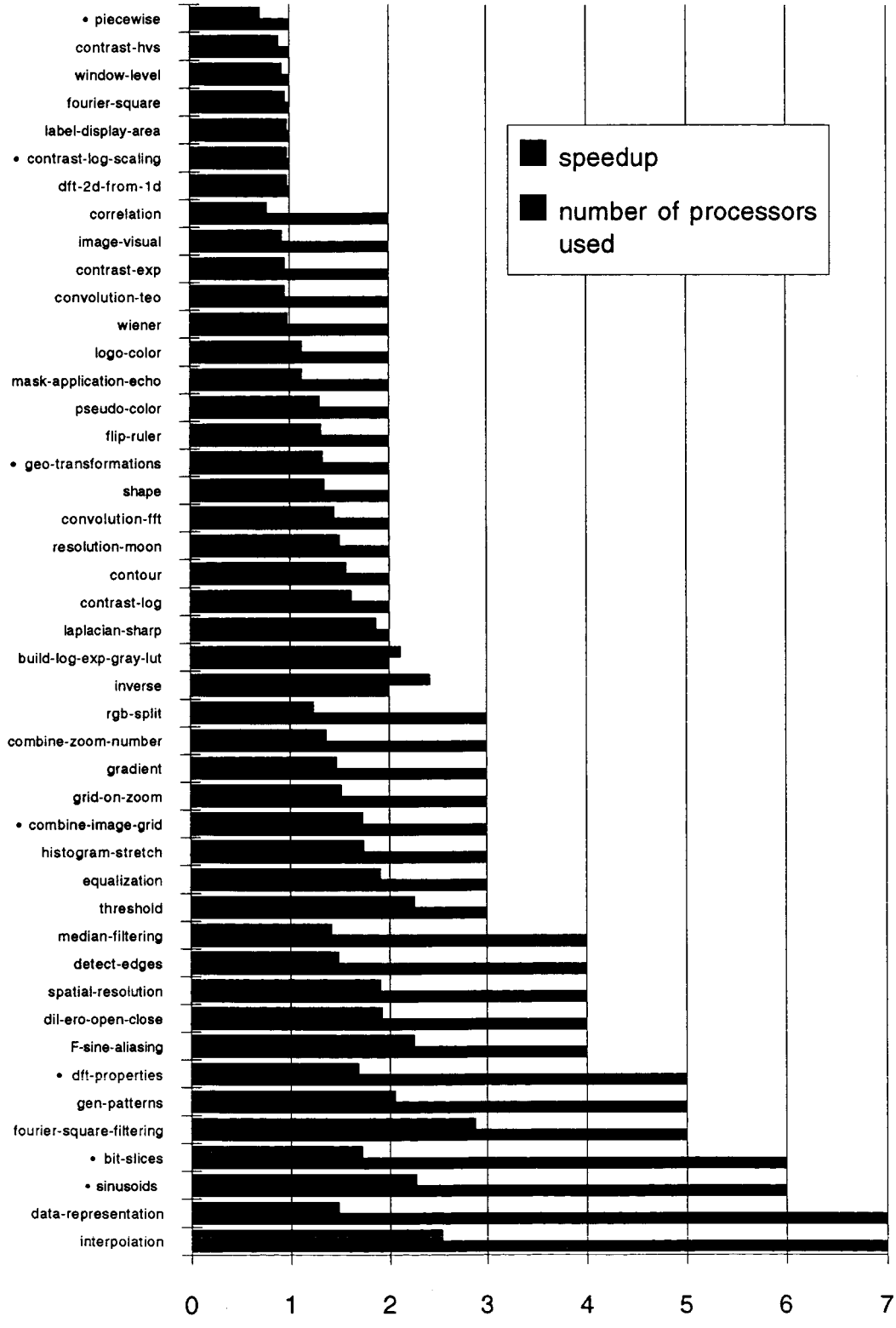

Figure 9: Speedup of the program graphs using default directives

4.3 Performance Study 2 - User Directed Scheduling

The second study investigates how well the environment responds to the user's scheduling directives. Multiple tests were executed as part of this study:

1. A program completion time preference test
2. A processor finish time preference test
3. A task ordering preference test
4. A task-to-processor assignment preference test

4.3.1 A Program Completion Time Preference Test

The goal of this test is to minimize program completion time. The default directives fulfill this goal. This is evidenced by the speedup of 1.4 obtained on the program graphs in performance study 1. Additional speedups were also obtained using the default directives on a set of computer-based scientific experiments. This data is presented in Section 4.4.

4.3.2 A Processor Finish Time Preference Test

The goal of this test is to prefer the finish time of one processor be at least twice the finish time of another processor. The processor that finishes early can be used for other computing tasks the user has in mind. For the test, two directives are used in addition to the second through fifth default directives. The first new directive requires that the environment only schedule tasks on the processors oddvar and norge. The second new directive requests that the finish time of the processor oddvar be at least twice that of the processor norge.

Figure 10 shows the results of the test. Notice that the finish time of processor oddvar is always at least twice the finish time of processor norge as the user requested.[12]

[12]Note that the reported results are execution times. Therefore they show the accuracy of the performance prediction tool as well as the quality of the scheduler. That is, the scheduler might fulfill the user's directives, but if its performance prediction information was incorrect, the execution results would most likely not fulfill the user's directives.

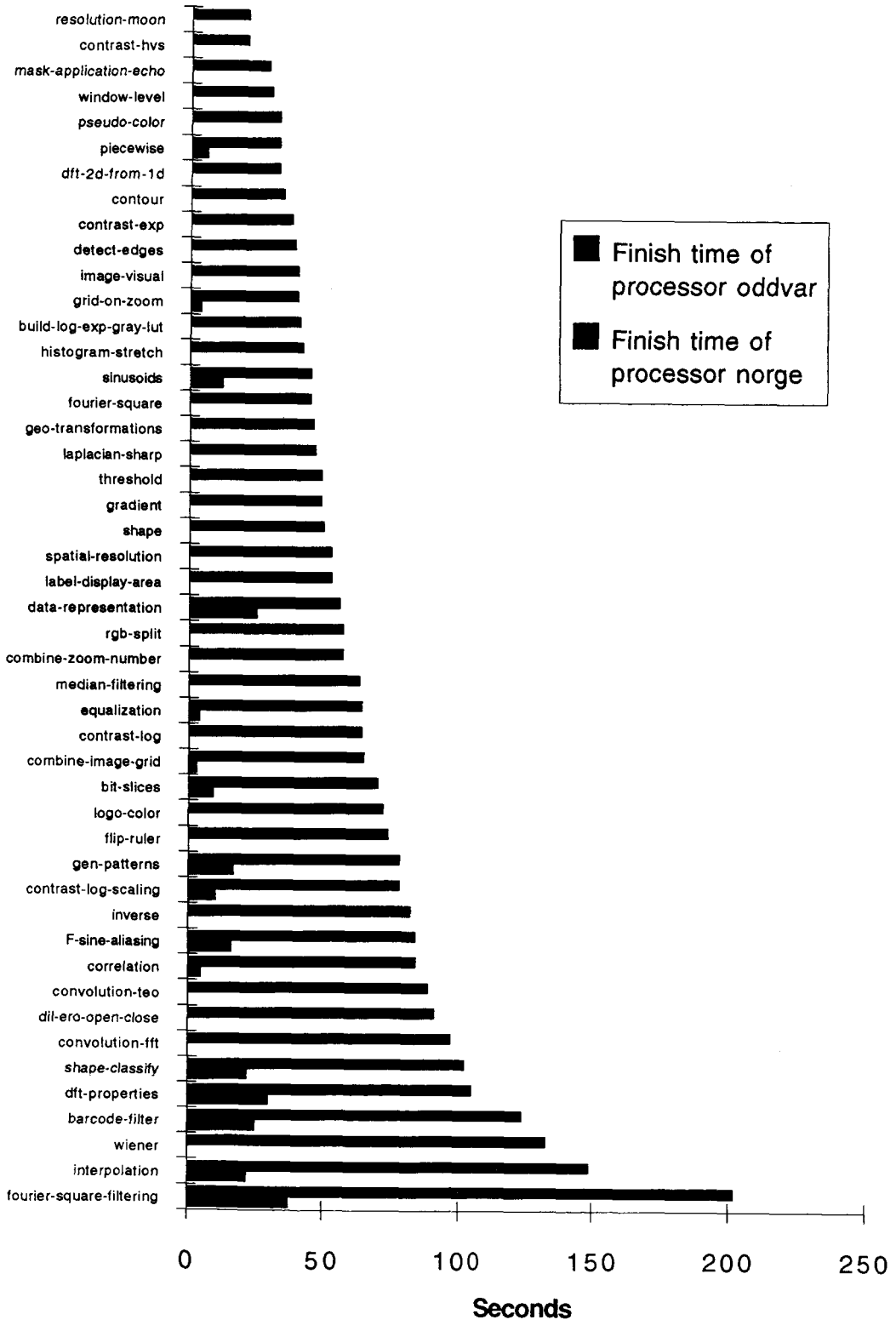

Figure 10: Processor finish time directive results

4.3.3 A Task Ordering Preference Test

The goal of this test is to prefer a particular ordering of task outputs. For the task ordering preference test, multiple directives are added in addition to the default directives. Each directive adds a dependency between a pair of output tasks to achieve this goal.

For the test, the output tasks of each DIP course program graph were identified, a random ordering of the tasks was generated and this ordering was preferred. The environment ordered the output tasks of all tested program graphs as requested. Table 1 presents a sample of the results of the test. The first column of the table lists the name of the tested program graph. The remaining columns lists the finish time in seconds of the tasks the user preferred to be output first, second, third, etc. Notice that the tasks are output in the order the user requested.

Table 1: Task ordering directive results

Program Graph	Finish Time 1st Task	Finish Time 2nd Task	Finish Time 3rd Task	Finish Time 4th Task	Finish Time 5th Task
combine-zoom-number	20 sec.	29 sec.	29 sec.	42 sec.	43 sec.
detect-edges	19 sec.	19 sec.	20 sec.	28 sec.	29 sec.
label-display-area	9 sec.	16 sec.	34 sec.	34 sec.	51 sec.
spatial-resolution	15 sec.	17 sec.	18 sec.	19 sec.	27 sec.

4.3.4 A Task-to-Processor Assignment Preference Test

The goal of this test is to prefer a particular task-to-processor assignment. For the test, a single directive is added in addition to the default directives. The new directive prefers that all "Display Image" tasks execute on the processor willow.

The DIP course program graphs were scheduled and executed using these directives and all "Display Image" tasks of each program graph were scheduled on the processor willow. Figure 11 shows an example result schedule. Notice that all "Display Image" tasks are scheduled on willow. Notice also that the default directives work in concert with the task-to-processor assignment directive to cause the tasks to be scheduled on multiple processors in parallel, reducing program completion time.

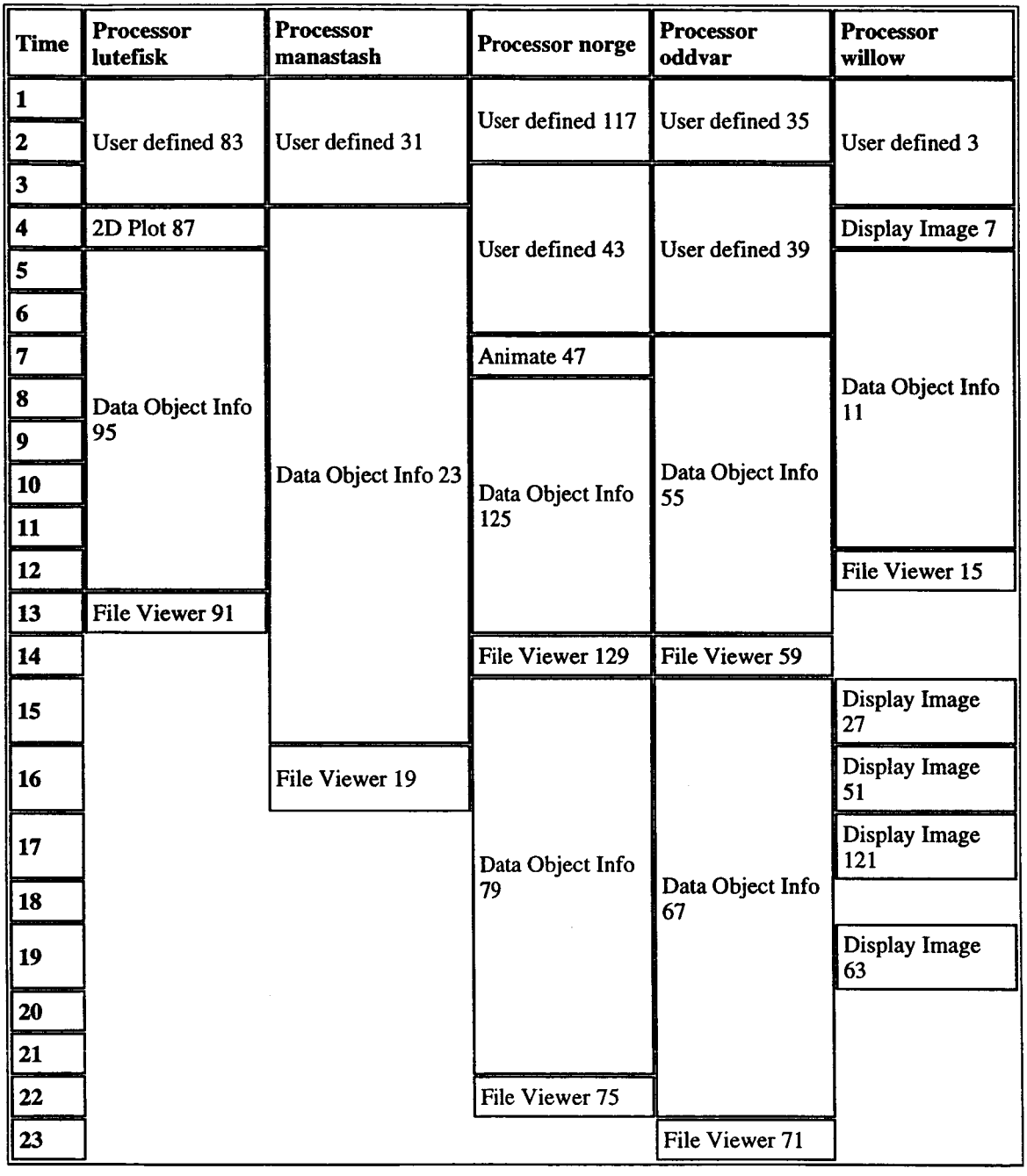

Figure 11: A schedule created using a directive which prefers all "Display Image" tasks be scheduled on the processor willow.

4.4 Performance study 3 - Computer-Based Scientific Experimentation

The third study explores the performance of the environment on a set of computer-based scientific experiments. Experiments are created using the program graphs of the DIP course. For each experiment, an input task and non-input task are randomly chosen. In the experiment, four different versions of both the input and non-input tasks were tested. Figure 12 presents the speedup of computer-based scientific experiments. The result data is presented in the same manner as the speedup data in Figure 9. The average speedup is 3.4 on an average of 5.5 scheduler processors. Notice the significant increase in speedup of these graphs. This is because experimentation creates many independent execution paths.

Figure 13 presents a comparison of the experiment creation algorithm described in this report to the simple method of replicating the entire program graph for each experimental substitution used by Nimrod[4]. This graph shows the finish time of the experiment created with the experiment creation algorithm described in this report along with an estimate of the finish time of an experiment created with the simple method. The estimated finish time for the simple method is calculated by multiplying the time to run the original program graph on a single processor by the number of replications (i.e. in this case, 4 x 4 = 16 replications). This is the time required to execute the experiment on one processor. This time is divided by the number of scheduled processors used when scheduling the experiment created by the experiment creation algorithm described in this report. This provides the optimal finish time possible for the simple method. Notice that because the experiment creation algorithm described in this report reduces the workload required to create experimental results, its finish time is usually less than the optimal finish time of the simple method.

4.5 User surveys

A survey was given to potential users of the scientific computing environment in order to assess its usefulness. Three vision researchers and a geologist who works on remote sensing applications saw a demostration of the environment and completed a survey. In summary, the users felt the environment would be useful for their computer-based scientific research work. Specifically, in response to the question, "If you were running programs on a shared distributed network of workstations, is the scheduler a tool you would find useful for your scientific research work?" the geologist responded "Yes - this would be useful now in the remote sensing lab as many users attempt to share a network of workstations.". The survey also tried to assess how familiar the scientists were with the tools used in the environment. Most had used a visual programming environment but not the relational database language SQL. They did not think that this would be a hinderance to learning the scheduling language, however. In fact, in response to the question, "Is the scheduling language easy to learn and

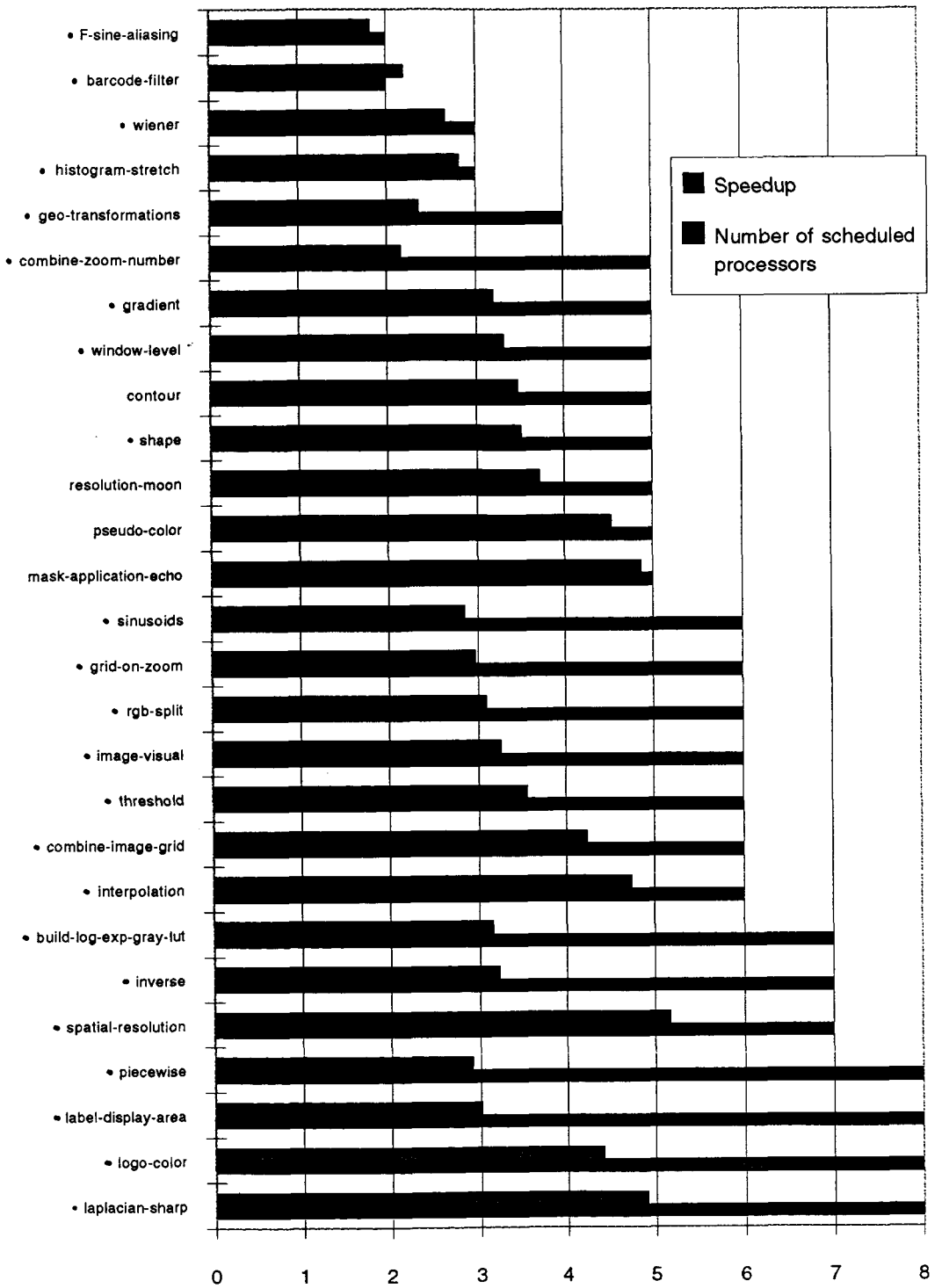

Figure 12: Speedup of experiments

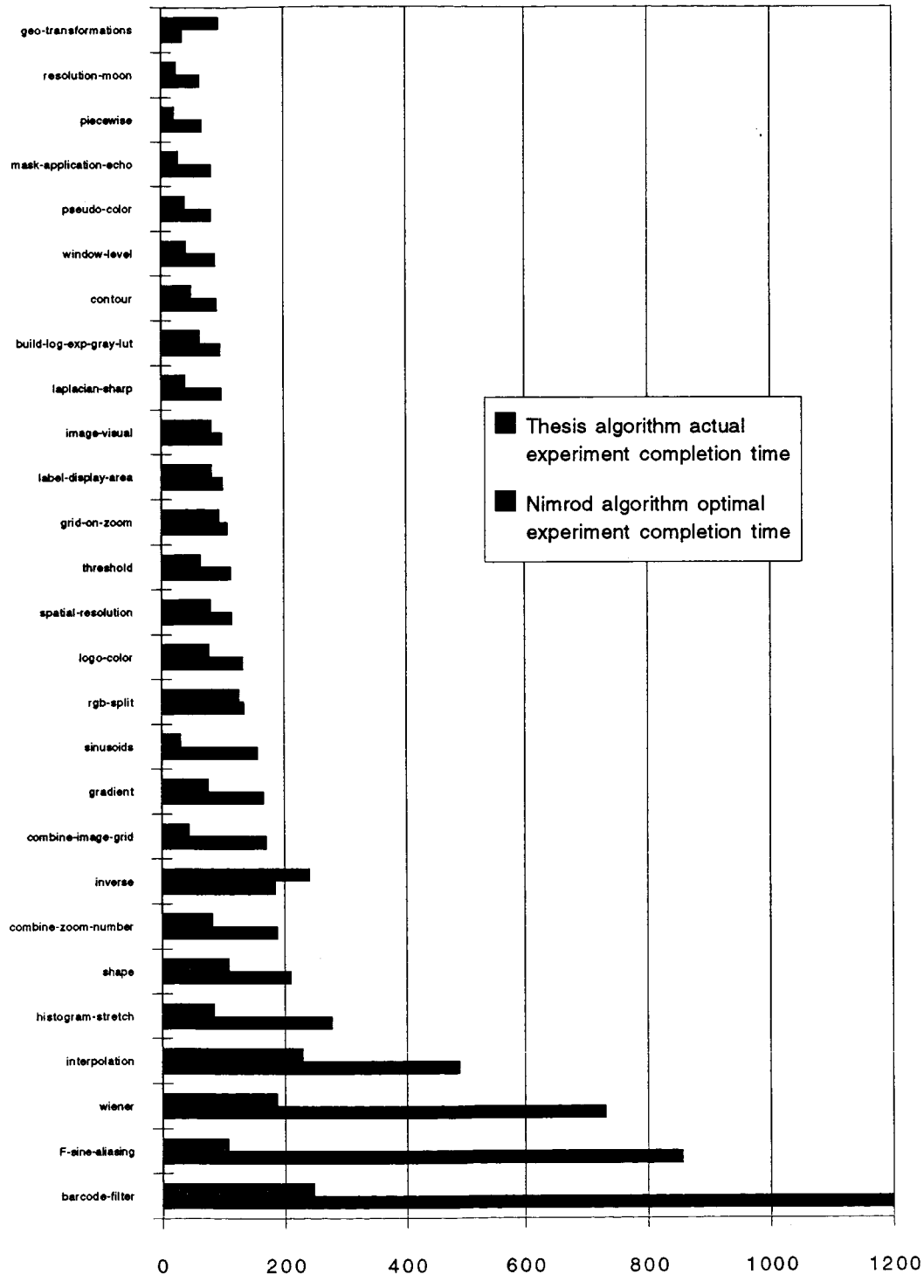

Figure 13: Comparison of experiment creation techniques

use?" all responded affirmatively. The users were also asked to order the usefulness of a collection of specific directives. The following list summarizes the user's choices:

- Minimizing program completion time: **1**
- Controlling task/processor assignments: **2**
- Output ordering: **3**
- Controlling processor utilization: **4**
- Time-related directives (after 3:00, before 6:00): **5**

Finally, the users were asked: "Is the support for computer-based scientific experimentation, a feature you would find useful for your scientific research work?" and most scientists responded positively with specific examples of research problems which would benefit from automatic experiment creation and execution. The full results of the geologist's survey is presented in Appendix A.

4.6 Summary

This completes the performance study of the environment. In summary, the study has shown:

- Using the default directives, an average speedup of 1.4 on an average of 2.8 scheduled processors is achieved on the DIP course program graphs.
- The environment is responsive to the user's scheduling directives. In a variety of tests including a processor finish time test, task ordering test and task-to-processor assignment test the environment fulfilled the user's directives for all program graphs.
- The environment achieves very good performance on scientific experiments. An average speedup of 3.4 on an average of 5.5 scheduled processors is achieved. In addition, the experiment creation method presented in this report creates more efficient experiments than the simple method used by the Nimrod environment. On average, the experiments execute 2.1 times faster than an optimal execution of the experiments generated by the simple method.
- A survey was given to potential users of the scientific computing environment in order to assess its usefulness. In summary, the users felt the environment would be useful for their computer-based scientific research work.

5 Conclusions and Future Work

This report describes a computing environment which supports computer-based scientific research work. Key features include support for automatic distributed scheduling and execution and computer-based scientific experimentation. A new flexible and extensible scheduling technique that is responsive to a user's scheduling directives, such as the ordering of program results and the specification of task assignments and processor utilization levels, is presented. An easy-to-use constraint language for specifying scheduling directives, based on the relational database query language SQL, is described along with a search-based algorithm for fulfilling these directives. A set of performance studies show that the environment can schedule and execute program graphs on a network of workstations as the user requests. An algorithm for automatically generating scientific experiments is presented. Experiments provide a concise method of specifying a large collection of parameterized program executions. The environment achieved significant speedups when executing experiments; for a large collection of scientific experiments an average speedup of 3.4 on an average of 5.5 scheduled processors was obtained.

Future work on the environment could consist of a high-performance implementation of the scheduler and extensions to support other types of parallelism. A more efficient implementation of the scheduler would allow the environment to quickly find solutions to very complex directives. Ideas for a more efficient implementation include using an imperative programming language, parallelism and incremental user directive calculations. Also, in addition to data-flow parallelism, the environment could be extended to support operator, pipeline and loop parallelism. The performance prediction tool would be extended to predict the performance of parallel and pipelined tasks. The scheduler and executor would need to be extended to handle these type of tasks as well.

References

[1] American national standard for information systems: Database language SQL. ANSI X3(135-1986), 1986. American National Standards Institute, New York.

[2] *CM/AVS User's Guide.* 1992. Thinking Machines Corporation, Cambridge, Massachusetts.

[3] A. Lansky and A. Philpot. AI-based planning for data analysis. *IEEE Expert*, 9(1):21–7, February 1994.

[4] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterized simulations using distributed workstations. In *Proceedings of the Fourth IEEE*

International Symposium on High Performance Distributed Computing, pages 112–121, August 1995.

[5] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the ACM SIGMETRICS Conference*, pages 267–78, May 1995.

[6] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):21–70, October 1986.

[7] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[8] T. Casavant and J. Kuhl. A taxonomy of scheduling in general purpose ditributed computing systems. *IEEE Transactions on Software Engineering.*, 14(2), 1988.

[9] R. Cigel, D. Carlson, and J. Maloney. Graphical executive language for engineering applications. In *MacNeal Schwendler World User's Conference*, 1992.

[10] S. Cross and E. Walker. Applying knowledge based planning and scheduling to crisis action planning. In M. Zweben and M. Fox, editors, *Intelligent Scheduling.* Morgan Kaufmann Publishers, 1994.

[11] C. J. Date. *A Guide to the SQL Standard.* Addison-Wesley, second edition, 1989.

[12] H. El-Rewini, T. Lewis, and H. Ali. *Task scheduling in parallel and distributed systems.* Prentice Hall, 1994.

[13] M. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan-Kaufmann, 1987.

[14] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[15] Nabil I. Hachem, Ke Qiu, Michael Gennert, and Matthew Ward. Managing derived data in the Gaea scientific DBMS. In *Proceeding of the Nineteenth Very Large Data Base Conference*, 1993.

[16] M. Jampel, E. Freuder, and M. Maher, editors. *Over-Constrained Systems.* Springer, 1996.

[17] M. Johnston. SPIKE: AI scheduling for NASA's Hubble space telescope. In *Proceedings 6th IEEE Conference on AI Applications*, pages 184–190, 1990.

[18] N. M. Short Jr. and L. Dickens. Automatic generation of products from terabyte-size geographical information systems using planning and scheduling. *International Journal of Geographical Information Systems*, 9(1):47–65, Jan.-Feb. 1995.

[19] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[20] P. Ma, E. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. C-31(1):41–7, January 1982.

[21] B. Myers, D. Guise, R. Dannenberg, B.Vander Zanden, D. Kosbie, P. Marchal, and E. Pervin. Comperhensive support for graphical, highly interactive user-interfaces: The garnet user interface development environment. *IEEE Computer*, 23(11):71–85, November 1990.

[22] G. Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH 1985 Conference Proceedings*, pages 235–243, July 1985.

[23] J. R. Rasure and C. S. Williams. An integrated data-flow visual language and software development environment. *Journal of Visual Languages and Computing*, 2(3):217–246, 1991.

[24] S. Ridlon. A software framework for enabling multidisciplinary analysis and optimization. In *6th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, September 1996.

[25] L. Shapiro and R. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:504–519, 1981.

[26] Linda G. Shapiro, Steven L. Tanimoto, James F. Brinkley, James P. Ahrens, Rex M. Jakobovits, and Lara M. Lewis. A visual database system for data and experiment management in model-based computer vision. In *Proceedings of the Second CAD-Based Vision Workshop*, pages 64–72, February 1994.

[27] Terence R. Smith, Jianwen Su, Divyakant Agrawal, and Amr El Abbadi. MDBS: A modeling and database system to support research in the earth sciences. In *Proceedings of the Workshop on Advances in Data Management for the Scientist and Engineer*, pages 90–99, February 1993.

[28] C. Upson, T. Faulhaber Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[29] M. Wu and D. Gajski. A programming aid for hypercube architectures. *Journal of Supercomputing*, 2(3):349–372, November 1988.

[30] M. Zweben, E. Davis, B. Daun, and M. Deale. Iterative repair for scheduling and rescheduling. *IEEE Systems, Man and Cybernetics*, 23(6):1588–96, Nov.-Dec. 1993.

A A Survey of Users of the Scientific Computing Environment

A.1 The survey results of Milton Smith, geologist, member of the University of Washington EOS Amazon project team

A.1.1 Requirements

Will a computing environment which fulfills the stated requirements (i.e. exploratory program creation, high-performance program execution, responsive to scheduling directives) be useful to you in your scientific research work?

Yes - it will assist in utilizing the computing resources available in the Remote Sensing Lab.

A.1.2 Test programs

Are the Digital Image Processing course program graphs representive of the types of programs you use in your scientific research work?

They are representative but not comprehensive. Research involves the continuous ingestion, evolution and development of new algorithms.

A.1.3 Visual programming environment

Is the visual program environment a tool you would find useful for expressing programs for your scientific research work?

Visualization is very important to communicating research results.

A.1.4 Scheduler

1. If you were running programs on a shared distributed network of workstations, is the scheduler a tool you would find useful for your scientific research work?

   Yes - this would be useful now in the remote sensing lab as many users attempt to share a network of workstations.

2. If you were running programs on a shared distributed network of workstations, which of the following directives do you think would be of useful to you?

   - Output ordering:
   - Controlling task/processor assignments: **3**
   - Controlling processor utilization: **4**
   - Minimizing program completion time: **1**
   - Time-related directives (after 3:00, before 6:00): **2**
   - Other directives you create:

3. Are you familiar with the database query language SQL?

   Yes to a limited extent.

4. Do you feel that the scheduling directive language would be easy to learn and use?

   Yes - no problem

5. Any other comments you have about the scheduler?

   None

A.1.5 Distributed program executor

Is the distributed program executor a tool you would find useful for your scientific research work?

Yes - it makes sense in terms of our distributed computing resources.

A.1.6 Computer-based scientific experimentation

1. Is the support for computer-based scientific experimentation, a feature you would find useful for your scientific research work?

   This is definitely the wave of the future. We are interested.

2. Any other comments about the environment's support for computer-based scientific experimentation?

   Make it easy for the user community to take responsibility for its evolution. Simple modular interfaces that allow expansion of capabilities.

A.1.7 Improvements

Do you have any suggestions for improving any component of the environment so that it would be useful to you for your scientific research work?

Actually use it.