LA-UR- 97-473

CONF-970835--1

Title: A DISTRIBUTED COMPUTING ENVIRONMENT WITH SUPPORT FOR CONSTRAINT-BASED TASK SCHEDULING AND SCIENTIFIC EXPERIMENTATION

Author(s): James P. Ahrens
Linda G. Shapiro
Steven L. Tanimoto

Submitted to: Sixth IEEE International Symposium on High Performance Distributed Computing

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos
NATIONAL LABORATORY

Form No. 836 R5
ST 2629 10/91

CONF-970835--1

A Distributed Computing Environment with Support for Constraint-Based Task Scheduling and Scientific Experimentation

James P. Ahrens[1], Linda G. Shapiro, Steven L. Tanimoto
Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, WA 98195-2350

Abstract

This paper 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 constraints, 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 constraints, based on the relational database query language SQL, is described along with a search-based algorithm for fulfilling these constraints. A set of performance studies show that the environment can schedule and execute program graphs on a network of workstations as the user requests. A method for automatically generating computer-based scientific experiments is described. 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.

1 Introduction

Scientists utilize a variety of different software tools to support the various aspects of their computer-based scientific research work. Examples include tools for program creation and execution and for data storage and retrieval. An *environment* is a collection of integrated tools that support an activity such as computer programming. Recent research has focused on creating integrated computing environments to support scientific research work. The purpose of these environments is two-fold: to provide a computer-based framework for scientific research activities and to provide an abstraction of the details of the scientist's computer system. If the framework is useful and accurate, then scientists do not have to spend time supporting their own interaction framework. If the abstraction is effective, then scientists do not become bogged down in computer management details and are more productive. The goal of our work is to create a distributed computing environment with these properties. This work is guided by the computing requirements of geologists working on complex remote sensing problems.

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

[1]Author's current address is Los Alamos National Laboratory, Mail Stop – B287, Los Alamos, NM 87545.

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). An integral part of this project is obtaining, manipulating, and understanding satellite images of these regions. We interviewed the members of this project 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: (1.) the computing environment should facilitate the scientist's exploration of different algorithmic solutions, (2.) results should be returned as quickly as possible, (3.) the environment should schedule and execute algorithms based on the scientist's constraints on resource utilization and algorithm execution, (4.) the environment's interface should provide support for non-expert users, letting the scientist specify a high-level description of his algorithms and requirements, (5.) the environment should record and organize the scientist's computer-based research work for later retrieval.

From these requirements we identified two keys features which are not well supported by existing software. These areas are:

1. **Support for automated constraint-based distributed 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.

1.1 Structure of the Computing Environment

The components of the computing environment are:

- **Data-flow based visual programming environment** – The scientist uses a visual programming environment to construct his programs. Examples include languages such as IBM's Data Explorer, AVS[15] and Cantata/KHOROS[13]. A user's program is expressed graphically as dataflow-based program graph. Users can manipulate the program graph interactively by adding and deleting tasks. Users have access to a library of existing tasks that 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.

- **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 used by the performance prediction tool. The executor utilizes PVM for distributed computing services such as remote process creation and interprocessor communication.
- **Scheduler** – The scheduler automatically schedules a program graph on a network of workstations based on the scientist's constraints. The environment's constraint-based task scheduling algorithm is described in the next section.
- **Performance prediction** – Program performance prediction is necessary for efficient scheduling. The scheduling algorithm uses performance estimates to make scheduling decisions.
- **Scientific database** – A database is used to organize and store information about program graphs and results.

A data-flow diagram of the scientific computing environment is shown in Figure 1. The diagram shows the data-flow between the components of the environment with boxes representing operations and ovals representing data. Directed arrows define the flow of data through the environment.

Data input to the environment includes resource information, a program graph and the user's scheduling constraints. Available processors are specified initially by the environment's administrator. The program graph is specified using a visual programming environment. The user scheduling constraints are specified using a constraint-based scheduling language. The program graph, resources and performance data from previous runs are used by the automatic performance prediction tool to create a model of program execution and processor utilization. The scheduler inputs the resource information, the program graph, the user's scheduling constraints and performance estimate information. The scheduler outputs a schedule which fulfills the user's scheduling constraints. 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.

2 Related Work

This paper describes a computing environment which automates performance prediction, scheduling and distributed execution of a scientist's computational experiments. The scientist expresses his program using a data-flow-based program graph representation. Environments with related features include: Sarkar's scientific programming environment[14], Goedecke *et al*'s circuit simulation environment[9], Lee *et al*'s imaging environment[12] and Short *et al*'s satellite image processing environment[11].

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 [3]. El-Rewini, Lewis and Ali [6] also provides a useful introduction to task scheduling. Since most types of task scheduling problems are NP-complete, solution algorithms are

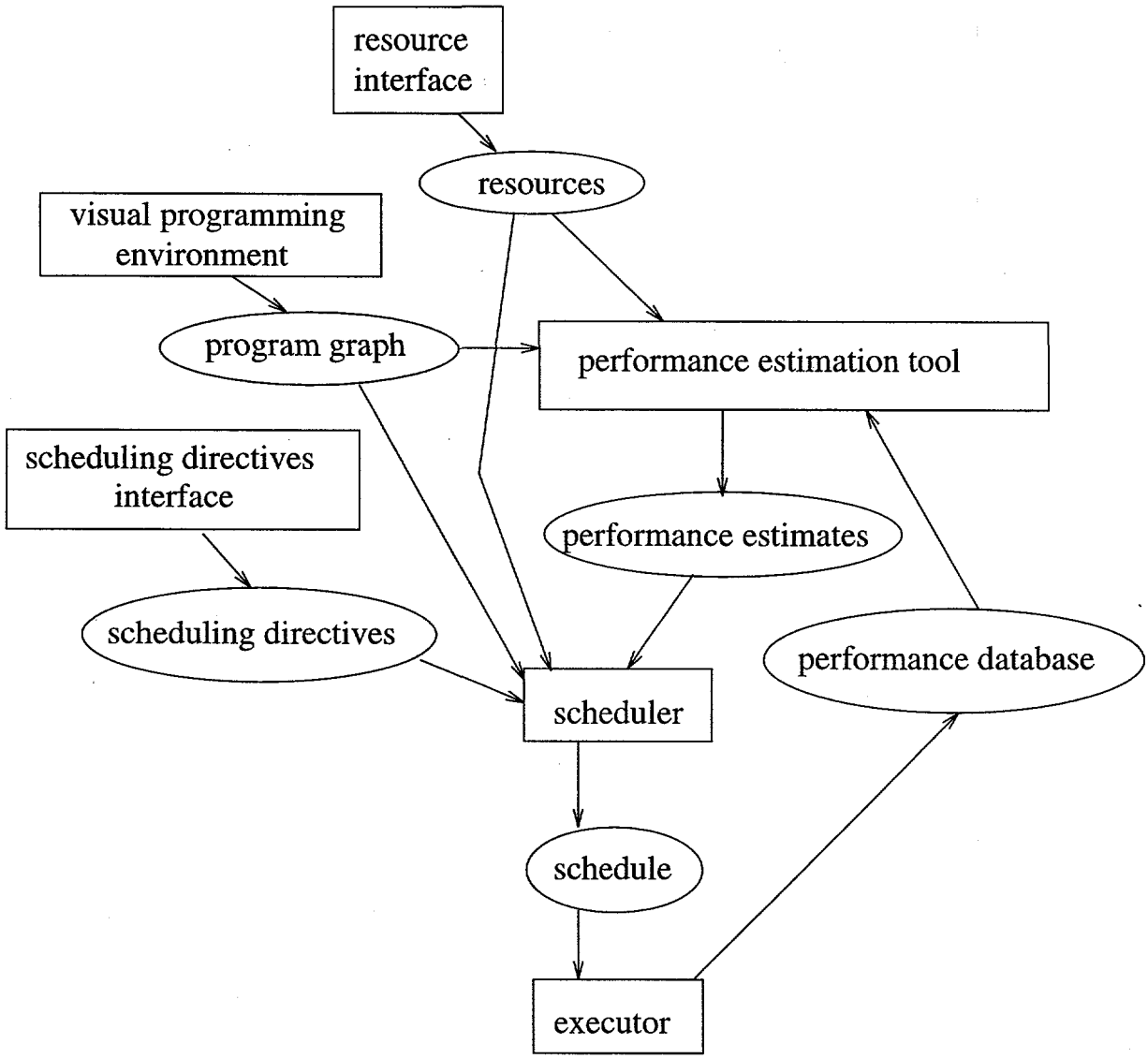


Figure 1: Structure of the scientific computing environment

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.

The goal of Sarkar's thesis [14] is to create portable, static and dynamic automatic scheduling algorithms for the functional programming language SISAL[7]. Goedecke's environment automatically dynamically schedules and executes a program graph created in Cantata/Khoros on a network of workstations[9]. Lee *et al*'s environment automatically schedules program graphs which consist of a linear chain of parallel tasks on a parallel machine with a mesh-interconnection network [12]. The parallel version of each task has a preferred distribution of its I/O data and this distribution may be different for two connected tasks. Their scheduler explores all possible different data layouts for the tasks and returns the set of layouts that optimize performance. Each of these environments is similar to ours in that they present a complete system that automates performance prediction, scheduling and execution. In addition, our environment offers a flexible scheduling mechanism, which can fulfill a user's scheduling constraints.

Scheduling is the process of assigning a set of jobs to a 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 [10], space shuttle maintenance [17], manufacturing [8] and defense logistics [4]. 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. There has been little previous work done on creating an AI-based task scheduler. The closest related work in this area is work by the members of the Intelligent Data Management Project led by Nicholas Short at NASA Goddard. Their group is working on a prototype environment which can process the massive datasets generated by satellites that are part of NASA's Earth Observing System [11]. The environment 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 environment supports automatic program creation using a planner to select and compose a set of these tasks into a program graph. In contrast, our environment schedules existing program graphs and fulfills the user's scheduling constraints. It allows 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.

3 Constraint-Based Task Scheduling

This section describes a scheduling algorithm that fulfills a user's scheduling constraints. First, a problem space representation for scheduling is described. This goal-oriented representation facilitates the

specification of scheduling constraints and is amenable to artificial-intelligence-based solution techniques including search and planning. Then a language for specifying scheduling constraints is defined. Finally, a search-based algorithm for determining a schedule is described.

3.1 A Problem Space Representation for Task Scheduling

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 three tasks, one which inputs an image, one which executes a Sobel edge detector and another which displays an image.

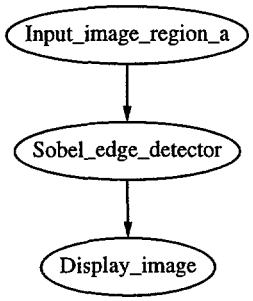


Figure 2: A program graph

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

This section describes a problem space representation for task scheduling. A problem space is defined as a set of states and the operators that perform transitions 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.

- **State** – 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. The attributes of the tasks, processors and task dependency entities are as follows:
  - **Task Entity** – a unique task id and name, the processor the task is assigned to and estimates of the task's execution, start and finish times.
  - **Processor Entity** – a unique processor id and name, estimates of the total running time of the tasks scheduled on this processor assuming no gaps or idle periods, an ordered list of tasks scheduled on this processor and the processor's CPU utilization.

- **Dependency Entity** – the time required to communicate this dependency data. If no communication is required then this value is 0.

- **Initial State** – The initial state contains a task entity for each task in the input data-flow program graph and 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.

- **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)* → *state.* [2] The result of executing the call, *schedule-task-to-processor(task-id, proc-id,original-state)* → *new-state* is that the task identified by *task-id* is scheduled on the processor identified by *proc-id.* The performance prediction tool fills in the attribute values of the newly scheduled task, processor and dependency entities.

- **Goal State** – The conditions required of a goal state are: (1.) Each task is scheduled to a processor. (2.) The task dependencies are respected by the schedule. That is, if a task is dependent upon another task for input, it will execute after that task has completed.

3.2 A Language for Expressing Scheduling Constraints

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, we 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 constraints* 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.

The scheduling language is an extension of SQL [1, 5] 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

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

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`.

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.* 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.

The first type of scheduling constraint 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 form 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 *relation-1* and *relation-2.* 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 - 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 form is:

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

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.

Relationship and Ordering Preferences The second type of scheduling constraint 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 similar 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 forms of the relationship and ordering preference functions are:

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

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

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 - 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)*:

- *opt-type* states whether to minimize or maximize the value function.
- *priority* is a measure of the importance of fulfilling this preference.
- *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: [3]

```
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.

3.3 A Search-based Scheduling Algorithm

In this section, we 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.

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

- **Successor Function** – The name and type of the successor function is: *successor(state)* $\rightarrow$ *set of states.* For the call *successor(state1)* the function executes the operator *schedule-task-to-processor* for all possible pairs of elements from the set of all tasks that can be executed by the set of all processors. 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.
- **Evaluation Function** – 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 name and type of the evaluation function is: *evaluation(state)* $\rightarrow$ *integer.* The evaluation function returns the prioritized sum of all scaled value preferences (i.e. g_{total} as defined below). Formally, for a set of $1 \ldots vp$ value 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.
  - s_i is the scaled preference value of preference i (s_i values are between 0 and 1 with 0 preferred).
  - g_i is the scaled prioritized value of preference i.

$$g_i = s_i * \frac{p_i}{p_{total}}$$

  - g_{total} is the sum of all the preferences' scaled prioritized values.

$$g_{total} = \sum_{i=1}^{vp} g_i$$

  Best-first search find an optimized goal state but not necessarily 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.
- **Goal Function** – The form of the goal function is: *goal(state)* $\rightarrow$ *boolean.* The goal function returns TRUE if all the tasks are scheduled and FALSE otherwise.

3.3.1 Understanding and Controlling the Scheduler's Behavior

This section helps the reader to understand how to control the scheduler's behavior. This section shows that the scheduler's search algorithm implements an A* search when a default scheduling constraint which prefers more scheduled tasks is used. Thus, insights about A* can be applied to the scheduler's search algorithm as well.

A* requires the search evaluation function, *F(current_state)*, be formulated in two part: *G(current_state)*, a function which measures the costs of getting from the initial state to the current state and *H(current_state)*, a function which estimates the cost of getting from the current state to the goal state. Assume that the first preference is the default scheduling constraint that prefers states with more scheduled tasks. The scheduler's search algorithm implements an A* search as shown by the following definition of G and H:

- $G(current\_state) = \sum_{i=2}^{vp} g_i$
- $H(current\_state) = g_1$
- $F(current\_state) = G(current\_state) + H(current\_state) = g_{total} = \sum_{i=1}^{vp} g_i$.

The function G expresses the cost of getting from the initial state to the current state as the result of the scheduler's evaluation function, except for the first constraint. Thus, it measures how well the current state fulfills the user's constraints. The function H expresses the cost of getting from the current state to a goal state as the number of remaining tasks to be scheduled multiplied by a scaling factor. The scaling factor, p_1 reduces the result of H, so that it does not dominate the G term.

Finding a Goal State Quickly – the Value of H Ideally, H would provide an accurate estimate of the distance to a goal state. If it did, the algorithm would proceed directly to a goal state without branching. A* has the property that if H never overestimates the cost to the goal state, then the optimal path to the goal, as determined by G, is found. Finding the optimal path implies finding the optimal goal state (i.e. the one which best fulfills the user's constraints) since G returns the results of the scheduler's evaluation function. H, however, does not always underestimate the distance to the goal state. The magnitude of H is controlled by the priority weight p_1. If p_1 is 0, then H is 0 and the search is a best-first search. As p_1 becomes larger, the value of H becomes larger and more important in the calculation of F. When H is very large, the search is very fast but the solution is not likely to be optimal.

Thus there is a time versus quality tradeoff inherent in the search algorithm. Users who want a schedule created quickly and are willing to accept a suboptimal solution should set priority p_1 to be much larger than other priorities $(p_2, ..., p_{vp})$. Users who are interested in an optimal solution should set p_1 to 0. The computing environment also allows the user to increase the value of p_1 manually while the scheduler is executing. Thus, the user can dynamically speed up a search that is not producing a schedule quickly enough.

The Search Space Landscape – the Value of G The function G defines the landscape of the search space, that is, the hills, plains and valleys of values the scheduler explores when searching for a goal state. These values can make the search for the goal state easy or difficult. When the scheduler searches for a goal state in a best-first manner, it explores lower elevations before higher elevations. Preference evaluation functions create these values, and therefore it is worth discussing how these functions affect the search. Preferences which have either of the following behaviors cause the scheduler to explore more states:

- **Increase in value as they approach a goal state**
  If a preference increases in value as it approaches a goal state, the scheduler is encouraged to explore all states in which the preference returns a smaller value first before exploring any state in which the preference returns a larger value. For example, the result value of a preference to minimize all processor finish times increases with each new scheduled task. In terms of the landscape of the search space, this is the case in which the goal states are at the peaks of hills.

- **Do not differentiate between states**

  If the preference value does not differentiate between states, it is difficult for the scheduler to find a path to the goal state, because all paths appear equally good. For example, suppose the user wants the finish time of processor norge to be exactly equal to half the finish time of processor oddvar. The user could express this relationship as follows:

```
assert value preference ( opt-type = minimize, priority = 10,

function = assert relationship preference (
    2 * (select proc-finish-time from proc where proc-name = ''norge'') =
        (select proc-finish-time from proc where proc-name = ''oddvar'')),

min = 0, max = 1)
```

  In terms of the landscape of the search space, this is the case where there are large plains in the search space.

All the different types of preferences (i.e. relationship, ordering and value preferences) can exhibit these behaviors. The user may be able to decrease the scheduler's search time by reformulating his preferences so that they do not exhibit these behaviors.

It is difficult to reformulate a preference which increases in value because in most cases such a preference is expressing an optimization to be performed by the search. It is easier to reformulate a preference so that it differentiates between states. For example, the finish time preference can be reformulated as follows:

```
assert value preference (opt-type = minimize, priority = 10,

function = absolute_value
   ((select proc-total-running-time from proc where proc-name = ''oddvar'') -
2 * (select proc-total-running-time from proc where proc-name = ''norge'')),

min = 0, max = (select sum(task-exec-time) from task) +
               (select sum(non-local-comm-time from dep)))
```

Notice that the first preference only differentiates states into two types; those in which the relationship holds and those in which it does not. The second preference differentiates states based on a computed distance of how well the relationship is fulfilled. Thus, when using the second preference, the search space contains more information, and the scheduler can find a path which proceeds to a goal state quickly.

4 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 the computing environment 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[4] in the program graph. 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 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 that 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 2.

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 3. All possible combinations of substitutions of input images and edge detection tasks are instantiated and executed as shown in Figure 4. The output images are labeled and stored in the database for later analysis.

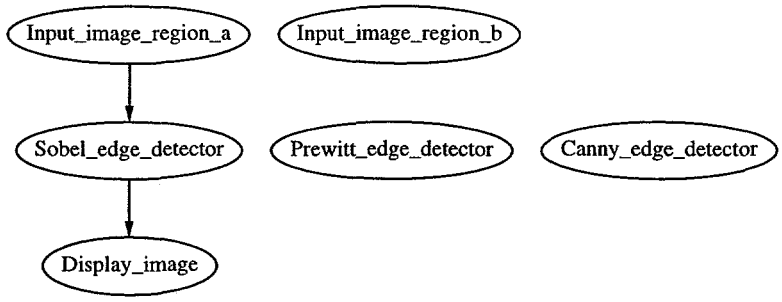


Figure 3: Substitutions for the experiment

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 4. This simple method requires more task executions than are necessary. For example, in Figure 4 notice that the Input_image_region_a task is

[4]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.

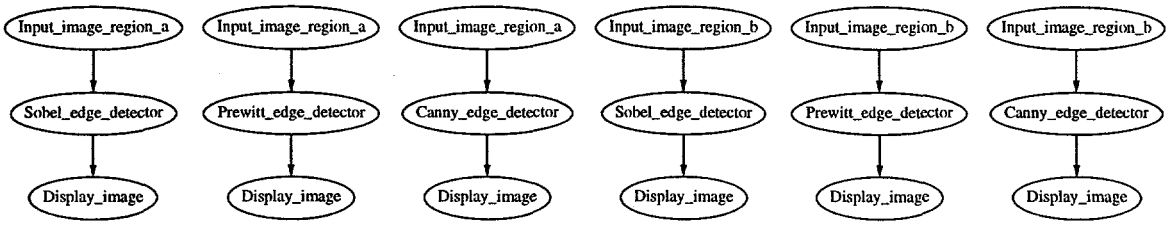


Figure 4: An instantiated experiment

executed three times although it is only necessary to execute it once. The computing environment 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.

A related environment, which executes experiments on a collection of distributed workstations in parallel, was created by D. Abramson *et al*[2]. The environment, Nimrod, allows a user to express a set of input parameters and data changes for a program. Nimrod creates experiments in a similar manner to the example shown in Figure 4. 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.

Our computing environment 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 replaced, replacing it with new code and then recompiling. After this process is complete, he can use the Nimrod environment to run experiments. In our computing environment, programs are represented as a collection of communicating 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 that perform the same function, such as edge detection.

Our computing environment also reduces the total amount of work required to execute an experiment. It uses an experiment creation algorithm that reuses task results whenever possible during an experiment. This algorithm allows scientists to obtain their experimental results faster than Nimrod's experiment creation algorithm, which replicates the entire program for each substitution. Nimrod 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.

Our computing environment allows user-directed scheduling of experiments. In the computing environment, experiments are expressed as program graphs. Therefore, the user can specify scheduling directives including task ordering, task-to-processor assignments and processor utilizations. The Nimrod environment allows its users limited control over the scheduling process. Users can prioritize the completion of particular experiments and can set a limit on the number of experiments concurrently assigned to a particular processor. Our computing environment's user-directed scheduling facility offers

a more general-purpose and extensible method of controlling the scheduling process than the limited set of commands provided by Nimrod.

5 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 constraints on a diverse collection of image processing program graphs. The second study examines the performance of the environment on computer-based scientific experiments. The third study investigates how well the environment responds to the user's scheduling constraints.

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. [5] 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.

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

The first default constraint 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 constraint 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 constraint directs the scheduler to prefer states with more scheduled tasks. In all tests, the priority value of this constraint was set to a value significantly greater than the other priorities. This constraint allows the search algorithm to make efficient progress. The next three constraints emulate Wu and Gajski's task scheduling algorithm[16]. 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 constraints, the environment executes the DIP course program graphs. Figure 5 shows the number of states explored by the scheduler for the program graphs. Notice that for most

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

graphs the environment explores less than five hundred states. [6] Thus, the scheduler, when using the default constraints, only needs to explore a small portion of the search space. The time required to execute the scheduler on these graphs ranged from a few seconds to, in the worst case, a few minutes.

Figure 6 presents the speedup achieved by the environment using the default constraints for the collection of program graphs. [7] It is important to study the speedup achieved by the environment to assess the performance of the default constraints. 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 6 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. [8] 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.

The second 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 7 presents the speedup of computer-based scientific experiments. The result data is presented in the same manner as the speedup data in Figure 6. 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.

The third study investigates how well the environment responds to the user's scheduling constraints. Three tests were run as part of the study:

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

The goal of the first test of the third study 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 constraints are used in addition to the second through fifth

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

[7] 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.

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

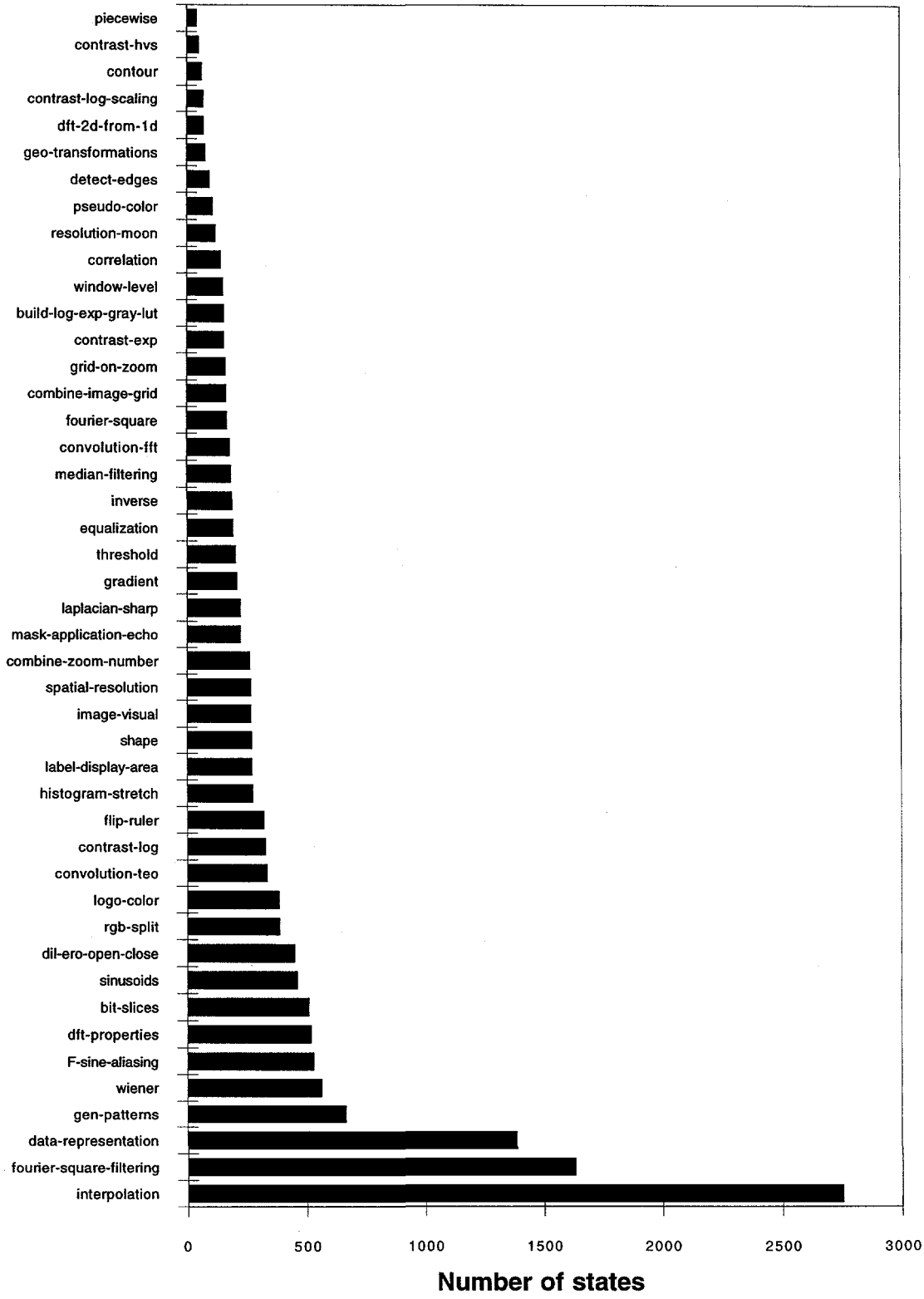


Figure 5: Number of states explored by the scheduler

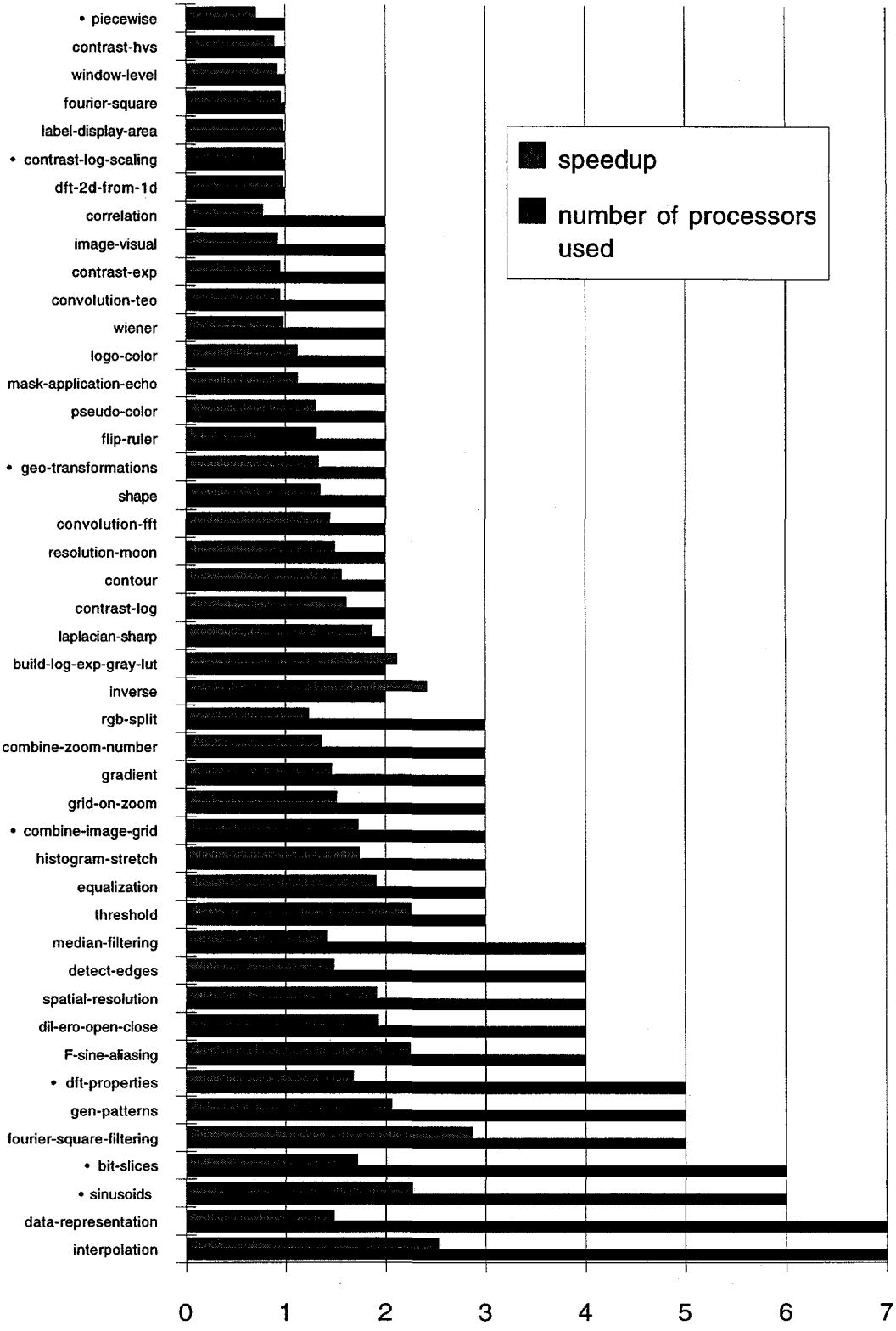


Figure 6: Speedup of the program graphs using default constraints

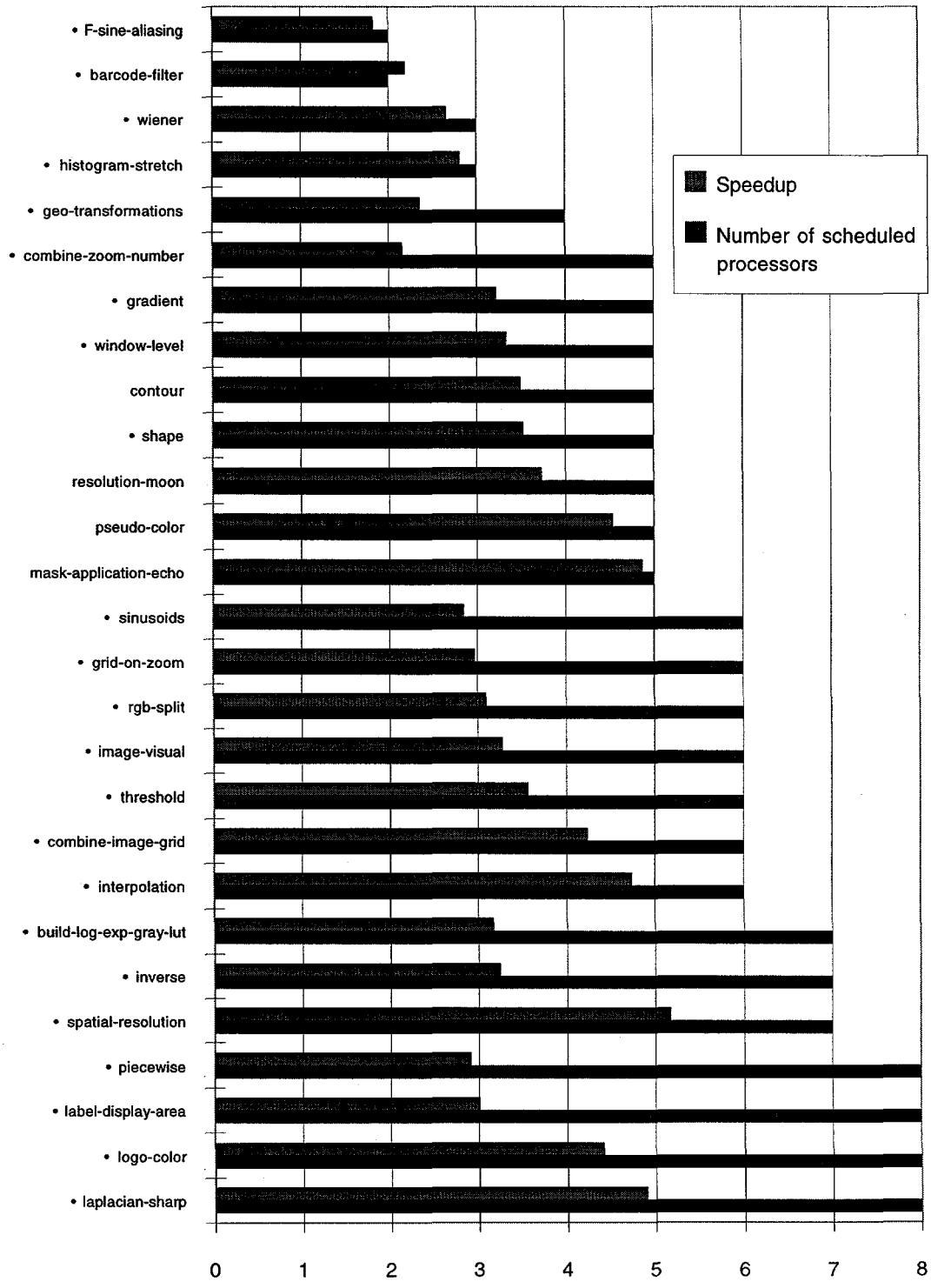


Figure 7: Speedup of experiments

default constraints. The first new constraint requires that the environment only schedule tasks on the processors oddvar and norge. The second new constraint requests that the finish time of the processor oddvar be at least twice that of the processor norge.

Figure 8 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.[9]

The goal of the second test of the third study is to prefer a particular ordering of task outputs. For the task ordering preference test, multiple constraints are added in addition to the default constraints. Each constraint 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 constraint 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.

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

The DIP course program graphs were scheduled and executed using these constraints and all "Display Image" tasks of each program graph were scheduled on the processor willow. Figure 9 shows an example result schedule. Notice that all "Display Image" tasks are scheduled on willow. Notice also that the default constraints work in concert with the task-to-processor assignment constraint to cause the tasks to be scheduled on multiple processors in parallel, reducing program completion time.

[9]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 constraints, but if its performance prediction information was incorrect, the execution results would most likely not fulfill the user's constraints.

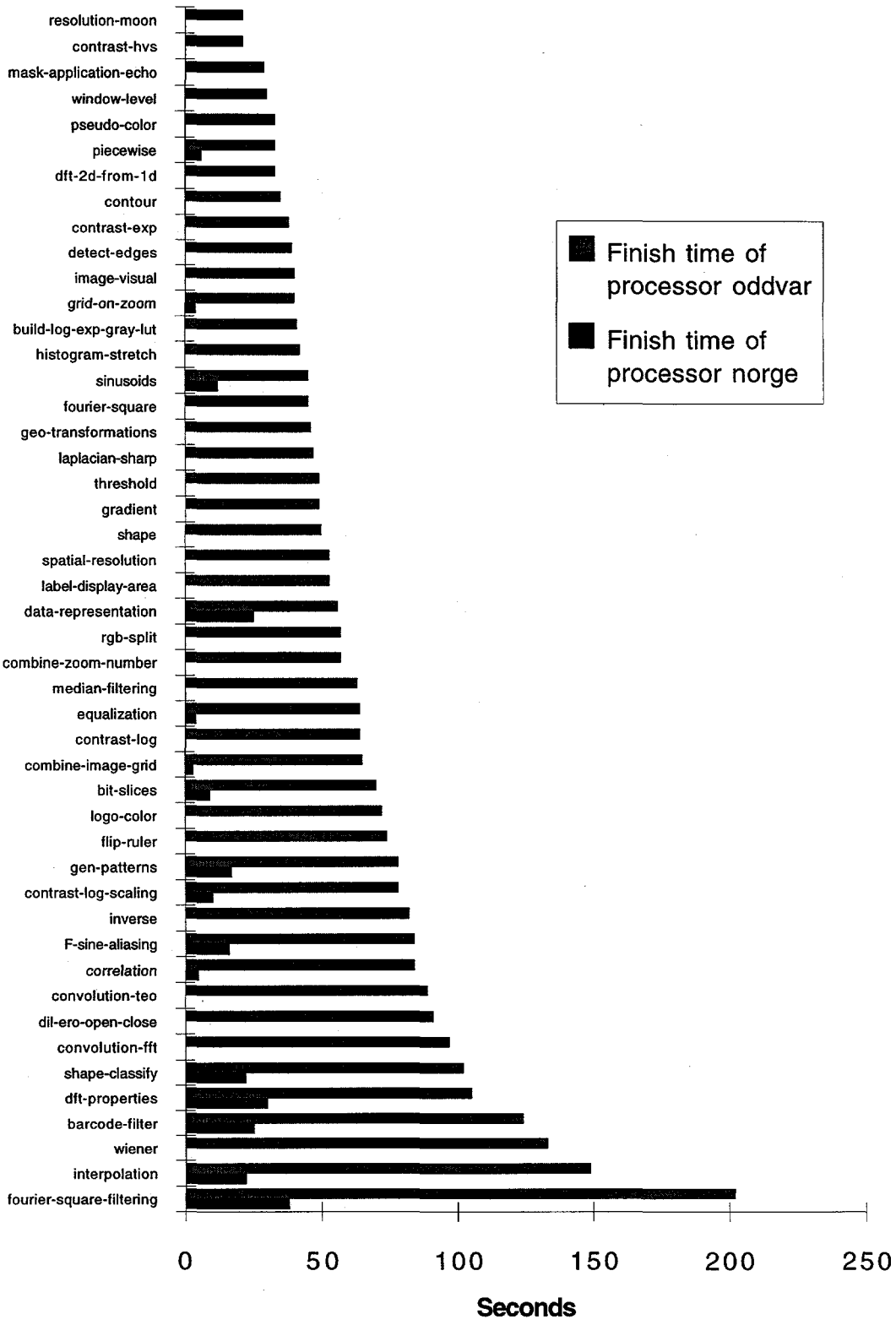
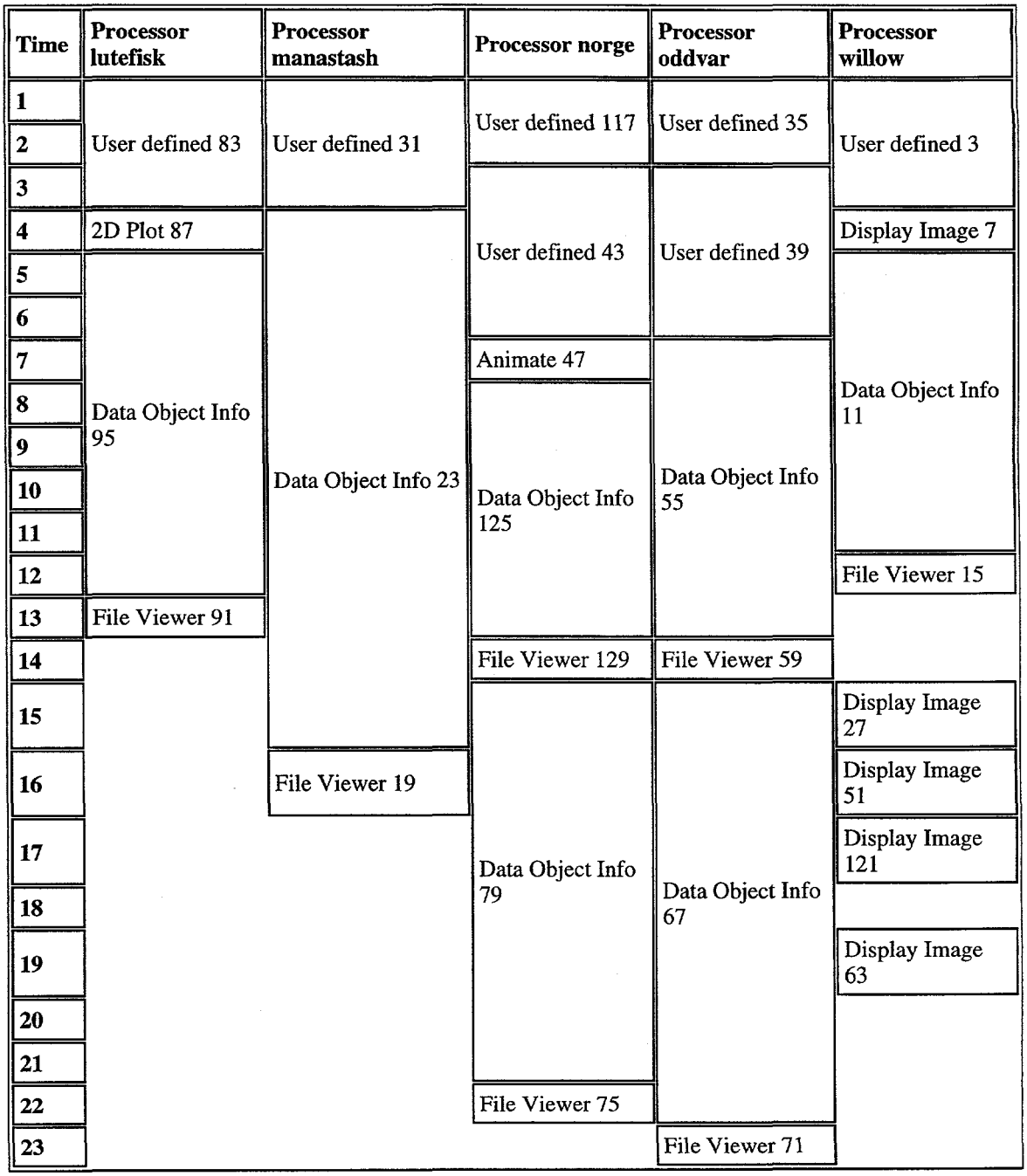


Figure 8: Processor finish time constraint results



Time	Processor lutefisk	Processor manastash	Processor norge	Processor oddvar	Processor willow
1	User defined 83	User defined 31	User defined 117	User defined 35	User defined 3
2					
3			User defined 43	User defined 39	
4	2D Plot 87	Data Object Info 23			Display Image 7
5	Data Object Info 95				Data Object Info 11
6					
7			Animate 47		
8			Data Object Info 125	Data Object Info 55	
9					
10					
11					
12					File Viewer 15
13	File Viewer 91				
14			File Viewer 129	File Viewer 59	
15			Data Object Info 79	Data Object Info 67	Display Image 27
16		File Viewer 19			Display Image 51
17					Display Image 121
18					
19					Display Image 63
20					
21					
22			File Viewer 75		
23				File Viewer 71	

Figure 9: A schedule created using a constraint which prefers all "Display Image" tasks be scheduled on the processor willow.

6 Conclusions

This paper describes a distributed computing environment with support for constraint-based task scheduling and computer-based scientific experimentation. A new flexible and extensible scheduling technique that is responsive to a user's scheduling constraints, such as the ordering of program results and the specification of task assignments and processor utilization levels, is presented. A set of performance studies show that the environment can schedule and execute program graphs on a network of workstations as the user requests. 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.

References

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

[2] 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.

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

[4] 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.

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

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

[7] J. McGraw et al. Sisal: Streams and iteration in a single assignment language – language reference manual – version 1.2. Technical Report M-146, LLNL, March 1985.

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

[9] M. Goedecke, S. Huss, and K. Morich. Automatic parallelization of the visual data-flow language cantata for efficient characterization of analog circuit behavior. In *Proceedings of the 11th IEEE International Symposium on Visual Languages*, pages 69–76, September 1995.

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

[11] 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.

[12] C. Lee, Y. F. Wang, and T. Yang. Static global scheduling for optimal computer vision and image processing operations on distributed-memory multiprocessors. Technical report, University of California at Santa Barbara, December 1994.

[13] 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.

[14] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[15] 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.

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

[17] 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.

DISCLAIMER