Translating an Existing Scientific Application from C to Dataparallel C

R. Jesse Chaney* Lawrence A. Crowl†

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 93-80-17

September 1995

Abstract

This report describes the translation of an existing sequential scientific program written the C programming language into the parallel programming language Dataparallel C. The resulting Dataparallel C program is able to use a network of workstations as though it were a single high-performance parallel computer. We describe the amount of effort required to translate the existing scientific program and the timing results of the translated program are compared to the sequential program. We conclude with some recommendations to parallel programming language designers, implementors of parallel programming languages, and application programmers.

*NSFNet: jchaney@research.cs.orst.edu, Phone: (503) 754-4497

†NSFNet: crowl@research.cs.orst.edu, Phone: (503) 737-2554

Contents

List of Figures

1 Introduction

With the increasing demand for computing power and the decreasing cost of parallel computing systems, we can expect to see not only a substantial number of new applications written for parallel computers, but also an increasing need to translate existing sequential applications to parallel systems.

The need to translate existing sequential applications to parallel systems is reinforced by the emerging technology of network computing. In network computing, applications coordinate the execution of programs on a network of standard, independently operating computer workstations. The resulting network can deliver performance comparable to a single parallel super-computer. Network computing is particularly rewarding because users may gain super-computer performance from their existing computer network, without purchasing additional hardware.

To explore the issues involved in translating an existing sequential scientific application to parallel computing environment, we parallelized a climate-driven vegetation model. Section 2 describes the program that we translated, its input, and its output.

The original program was written in C, and we chose to translate it to Dataparallel C [Hatcher and Quinn, 1991]. Dataparallel C is a parallel programming extension to the C programming language. Its extensions provide support for the single instruction multiple data (SIMD) model of parallel computation. We chose Dataparallel C because (1) Dataparallel C is based on C and therefore the majority of our program should not need to change, (2) Dataparallel C has an implementation that uses a network of workstations [Nedeljkovic and Quinn, 1992], we could explore the network computing model. Section 3 contains a overview of the data-parallel model of computation, the Dataparallel C language, and the implementation of Dataparallel C which we used.

Section 4 details the effort needed to translate the original program into Dataparallel C. Issues which arose during the translation can be divided into three classes: problems or limitations derived from the Dataparallel C language, problems with the current Dataparallel C compiler, and problems due to a structural mismatch between the original program and Dataparallel C. The single largest problem was an inability to use pre-existing sequential libraries.

In section 5, we describe the timing environment for both the sequential and the translated parallel program, the methods used for obtaining runtime, and describe the timing results.

Section 6 contains our concluding remarks — we address issues raised in the translation process: language specific, implementation specific, and general program writing comments. Section 7 lists some areas we would like to investigate in future research.

2 The Original Program

This exercise began with a working model for predicting steady state vegetation for current and possible future climatic conditions — the Mapped Atmospheric-Plant-Soil System (MAPSS) [Neilson]. MAPSS is a model used by environmental researchers in the United States Environmental Protection Agency's Corvallis Environmental Research Laboratory and the United States Forest Service's Global Environmental Protection Program in Corvallis, Oregon.[1] MAPSS is a central

[1]The research associated with MAPSS is supported by the United States Forestry Service, the United States Environmental Protection Agency, and the United States Parks Service.

portion of the United States Environmental Protection Agency's global climate change research program. MAPSS was not developed with parallel execution in mind, and hence serves as a test for the effort involved in translating a sequential scientific application to a parallel system.

2.1 How MAPSS Works

MAPSS predicts *potential natural* vegetation given a steady state climate. A steady state climate simply means that the climate does not change from year to year; each January always has the same amount of precipitation and the same temperature. The potential natural vegetation from MAPSS is what the model determines could be supported at the site if an *unbounded amount of time* was available under the given climate. All climate input variables remain constant and the model runs to a steady state solution. Figure 1 shows MAPSS predicted potential natural vegetation for the current climate.

The potential natural vegetation is distinct from the *current* vegetation. Whereas current vegetation can be seen on the ground, potential vegetation represents what "could" occur at a given site — it is driven strictly by climatic conditions. Current vegetation is easy to determine; one simply looks at it. On the other hand, potential natural vegetation for a site can be very difficult to determine because current vegetation is often affected by human influences, such as cultivation and grazing, and may not reflect the potential natural vegetation at all. Historical records or physical evidence may provide information as to what potential vegetation would occur at a site without human intervention, but this is not always the case. For example, it is known that for thousands of years the Native American inhabitants around Oregon's Willamette Valley periodically burned the valley. Historical records for the Willamette Valley do not predate the Native American inhabitants and physical evidence provides very little help in determining what vegetation would naturally occur in the Willamette Valley.

MAPSS iterates within the climatic conditions until an *equilibrium* is achieved between the given climate and a calculated vegetation. Equilibrium conditions are satisfied when the vegetation, as measured by leaf-area index[2] (LAI), for the current growing season matches the vegetation from the previous growing season. Each site is primed with LAI values for woody and grass vegetation at the beginning of the search for equilibrium. If the initial LAI values do not achieve an equilibrium with the climate, the the LAI values are changed and tested again. This repeats until an equilibrium is finally achieved.[3] For further information about the details of the MAPSS model readers should consult [Neilson; Neilson *et al.*, 1992; Neilson *et al.*, 1989; Daly *et al.*, 1993].

Each site in the input for MAPSS is processed independent of all other sites. MAPSS is a *one dimensional* model, the outputs for a given site do not depend on any neighboring sites. There is no need for information exchange between sites, no need for the computation to proceed with all

[2]Leaf-area index indicates the total surface area of the leaves above a certain area of the ground surface: $LAI = \frac{Total\ leaf\ area}{Ground\ area}$. Typically the units used for leaf area and ground area are identical, so LAI is a dimensionless measure of the amount of cover. A LAI value of 4 indicates that an area of ground is covered by 4 times as much leaf area. Higher numbers represent denser vegetation. A dense forest may have a tree LAI value as high as 14.0, an open grassland may have a grass LAI of less than 1.0 [Larcher, 73].

[3]Multiple equilibria, though possible, are not output — only the first will be found. A logic structure has been incorporated into MAPSS to avoid conflicts that may arise due to multiple equilibria. Later versions of MAPSS do investigate multiple solutions.

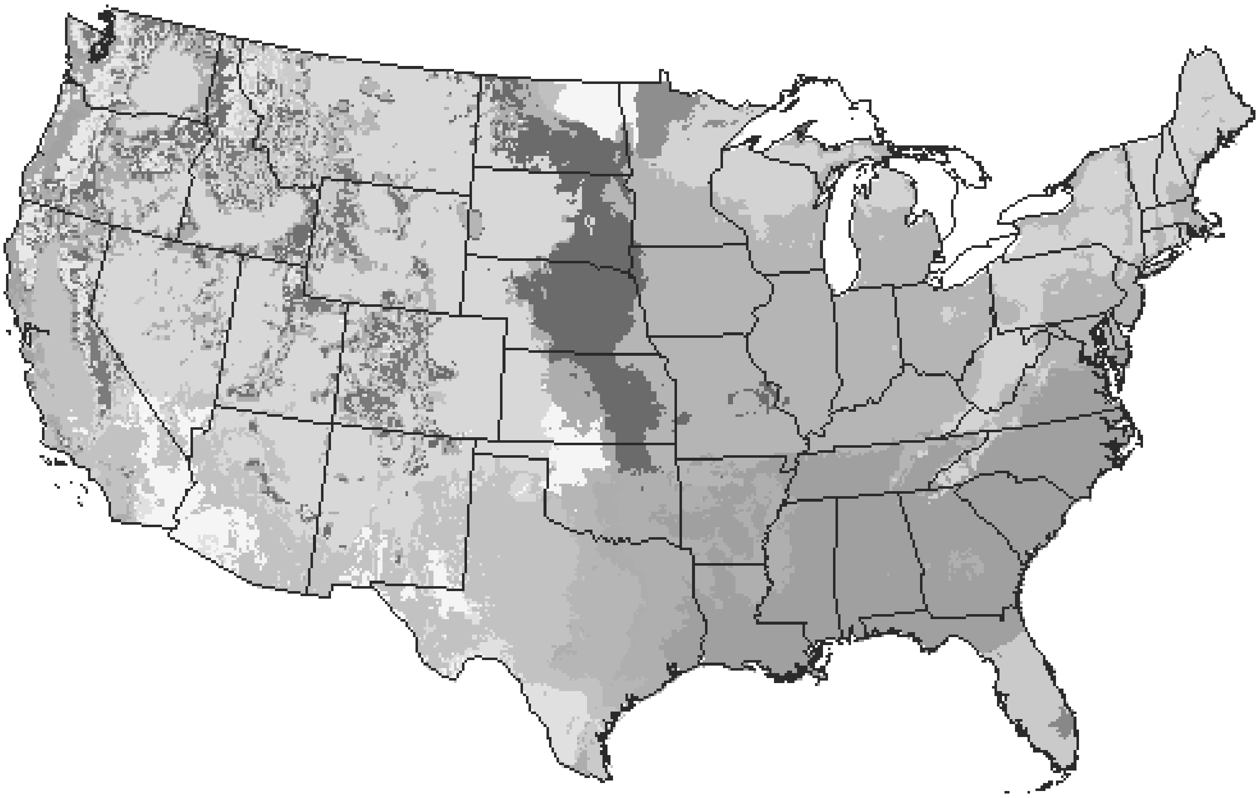

Figure 1: Potential natural vegetation as predicted by MAPSS.

sites in lock step. This makes it an especially appealing candidate for translation into a parallel programing system. It falls into the group of "embarrassingly parallel" applications.

Though MAPSS is highly parallel, it is not strictly SIMD. An important facet of MAPSS which was not given adequate emphasis during translation was the knowledge that sites will require a different number of iterations to achieve an equilibrium. Climate varies greatly between geographically distant sites. The number of iterations required to achieve equilibrium LAI values for the cool wet Pacific northwest of the United States will be different from the hot dry southwest. The non-uniform number of iterations required by a site will have a tremendous affect on how the work should be distributed and on how appropriate MAPSS is for data-parallel programming.

2.2 Input and Output

There are two sets of input data for MAPSS, one for the contiguous 48 states of the United States of America (hereafter we use USA to refer to only these states and exclude Alaska and Hawaii.) and one which covers the entire globe. The input files for the USA consist of 138,650 cells (295 rows by 470 columns). Each cell represents a 10 kilometer by 10 kilometer surface on a conical equal area Albers projection. Of the 138,650 cells, 77,857 contain actual data (56 percent). The remainder are masked out, being either land mass outside the United States or water. The global data set consists of 259,200 cells (360 rows by 720 columns). Each cell represents 1/2 degree by 1/2 degree latitude/longitude projection of the Earth's surface. Of the 259,200 cells, 62,483 contain actual data (24 percent); the remainder being masked out as water, the Arctic, or the Antarctic. The number of non-masked data cells in each data set is approximately equal. The individual climate variables for each cell consists of 12 monthly values for temperature, precipitation, vapor pressure, wind, and single values for elevation and mask; a total of 50 values for each dell.

Each data set contains six climate scenarios, one derived from the current climate and five derived from general circulation model (GCM) data. Each GCM represents a possible future climate based upon a doubling of the atmosphere's carbon-dioxide (CO_2) content. The five GCM data sets used with MAPSS are provided by:

- the Geophysical Fluid Dynamics Laboratory of NOAA (GFDL) [Manabe and Wetherald, 1987; Wetherald and Manabe, 1988],
- GFDL Q-flux (a modified GFDL model which includes a heat flux caused by ocean currents),
- the Goddard Institute for Space Studies (GISS) [Hansen *et al.*, 1983; Hansen *et al.*, 1988],
- the Oregon State University (OSU) [Schlesinger and Zhao, 1989], and
- the United Kingdom Meteorological Office (UKMO) [Mithcell and Warrilow, 1987].

Each GCM predicts changes to the individual climate variables. In general, the future climates as predicted from each GCM are drier and hotter in the USA. Some sections of the globe are affected in a positive manner by one or more of the GCMs, many are adversely affected.

The output for each cell consists of a predicted vegetation type (one of 34 classes), LAI values (one for each of grass, shrubs, and trees), and other variables related to the climate and vegetation, *e.g.* growing degree days, actual transpiration, and potential evapotranspiration[4] (PET). The output from MAPSS is studied to attempt to predict how vegetation may be affected by the climate changes predicted by each of the five GCMs.

2.3 Originally Developed in C

The MAPSS model was developed in the C programming language. The version to be translated began with over 10,000 lines of source in over 30 files. It had been processed through the UNIX[5] utility `lint` many times and showed no errors, warnings, or comments (except for a comment about an unused identifier from a system include file) [Johnson, 1977]. (See appendix A for a list of the options used with `lint` on the MAPSS C code.) In addition, the source was often run through the Free Software Foundation's C compiler, `gcc`, with options showing diagnostics and code analysis similar to those in `lint` [Stallman, 1992]. These resulted in warnings only for an unreferenced identifier used for `RCS` maintenance. (Consult appendix B for a list of the options used with `gcc` on the MAPSS C code.) The beginning code for MAPSS was therefore quite clean.

[4]Potential evapotranspiration is the sum of water evaporation from the soil and water transpiration through plant material, given an unlimited water supply [Larcher, 73]. There are several methods which may be used to estimate PET, MAPSS uses Brutsaert's interpretation of the Businger-Dyer model using Obukhov stability length [Marks, 1992].

[5]UNIX is registered trademark of AT&T Bell Laboratories.

MAPSS is not a stand alone system. It relies on third party libraries for data input and output. MAPSS uses a public domain image processing system called the Image Processing Workbench (IPW) [Frew, Jr., 1990; Longley and Marks, 1991; Longley *et al.*, 1992] for data input and a public domain geographic information system (GIS) called the Geographical Resources Analysis Support System (GRASS) [GRASS] for output. IPW is a large collection of programs and a C library developed by environmental researchers to ease creation and utilization of large quantities of geographic data (temperature, precipitation, cloud cover, *etc.*). An example of input data in IPW format is 12 months of average monthly temperature for the USA, a file of containing 138,650 values for each of the 12 months, approximately three mega-bytes. GRASS is a general purpose raster-based GIS developed by the United States Army Corps of Engineers, Construction Engineering Research Laboratory. GRASS allows a researcher to create, manipulate, and analyze spatial or geographic data. An example of a GRASS file is the vegetation classification of the lower 48 states of the United States (figure 1).

3 Dataparallel C

Dataparallel C is a *data-parallel* programming language which grew from the original C* data-parallel language (developed by Thinking Machines Corporation).[6] C* was developed as an extension to the C programing language [Kernighan and Ritchie, 1978] which could take advantage of the Connection Machine's processor array. Readers interested in more information on C* should consult [Rose and Steele, 1987].

Data-parallel languages express parallelism through the simultaneous application of a single operation to an array (or other structure) of data. The data-parallel model of computation has clear roots in the SIMD model of computation. Data-parallel programming extends the SIMD machine model in two ways: an unbounded (but finite) number of processors and a global name space. Hatcher and Quinn [Hatcher and Quinn, 1991] denote other characteristics of data-parallel languages: an imperative style, explicit parallelism, and a local view of computation.

Data-parallelism can be contrasted with *control-parallelism*, which uses the simultaneous execution of many different operations. It is often characterized by a small number of asynchronous processes, whose number does not scale well with the problem size. Data-parallelism is characterized by a large number of synchronous processes, the number of which can be easily scaled with the problem [Hatcher and Quinn, 1991, pp. 13–14].

3.1 A Virtual Parallel Computer

A Dataparallel C program runs on a *virtual computer* composed of a front-end uniprocessor connected to a configurable back-end SIMD computer. The front-end executes the sequential portions of a program, while the parallel sections execute on the SIMD back-end. The back-end computer is considered configurable because the number and topology of *virtual processors* may be selected as bests suits a problem (or sub-problem), independent of the physical hardware.[7] Different parallel

[6] C* and Connection Machine are registered trademarks of Thinking Machines Corporation.

[7] Though the size and topology may be selected without regard to the physical hardware, a program may execute significantly faster if the physical hardware is given consideration.

sections of a program may activate differently sized and configured back-end virtual computers. Only a single back-end configuration by be active at any one time. A single front-end can access any of several appropriately configured and scaled back-end computers.

The virtual SIMD computer for a Dataparallel C program is based upon three underlying concepts.

- **An unbounded number of virtual processors.** The virtual machine may be thought of as though there are as many virtual processors as are necessary or efficient for program abstraction. A virtual processor is intended to represent what a programmer believes is a basic unit of computation on a collection of data. The number and topology of virtual processors may be defined that is appropriate for each portion of a problem.
- **The synchronous execution of a single instruction stream.** Semantically, the same program statement is being executed on the entire set of active virtual processors at each time step. A programmer using Dataparallel C develops parallel programs using a SIMD model of computation.
- **A global name space.** The execution stream of one virtual processor may reference a variable on another virtual processor. Virtual processor interaction is done though expressions in the language. The reference may be done without regard for the physical location of the virtual processor and without explicit message passing by the programmer.

3.2 The Dataparallel C Language

Dataparallel C is a SIMD extension of the C programming language. It is a superset of the C programming language, save that it does not support the `goto` statement from within parallel code. Its syntax is the same as C, with some features and reserved words added. Dataparallel C adds data definition constructs to describe the memory layout of the virtual processors in a back-end computer and control constructs to explicitly define sections of code which activate back-end virtual processors.

```
domain Circles {
    int     x;
    int     y;
    float   radius;
    float   circumference;
    float   distance;
    float   area;
    float   compute_area();
};
```

Figure 2: A simple Dataparallel C virtual processor domain declaration.

Virtual Processor Description

A virtual processor's memory layout is described using a `domain` declaration, which is very similar to a C `struct` declaration (see figure 2). In addition to members of standard C types, the members of a domain may include *member functions* (figure 2, line 8). Instances of a domain are defined using the array syntax of C (see figure 3). Each element in the virtual processor array defined in figure 3 represents an instance of a virtual processor as described by the memory layout in figure 2.

Dataparallel C automatically partitions the virtual processors among the physical processors. A programmer is not required to find an efficient distribution of the data among the physical processors. It is likely that more than one virtual processor will be mapped to a single physical processor. The underlying Dataparallel C system may create loops on the physical processors to emulate the requested number of virtual processors. The translation of virtual processor to physical processor is transparent to the programmer.

```
#define NUM_COLS 1000
#define NUM_ROWS 1000
domain Circles circles[NUM_COLS][NUM_ROWS];
```

Figure 3: Declaration of a Dataparallel C back-end SIMD machine.

Activation of a Back-end Virtual Computer

A set of virtual processors for a back-end virtual machine is activated by the *domain select* statement (see figure 4, line 4). The `domain` statement in Dataparallel C is similar to the the `with` statement from Pascal — it allows the components of a virtual processor to be accessed using member names only. Upon activation, all virtual processors within a domain, semantically, perform operations in lock step. In addition to activating a virtual machine with domain select statement, virtual processors may be activated by calling a domain member function (figure 4, line 22). Calling a member function activates all the virtual processors which contain that function, exactly as a domain select would do.

The domain select statement and domain member function calls clearly divide sections of a Dataparallel C program into sequential execution and parallel execution. Whenever a domain select is active or a member function is called, a back-end virtual SIMD machine is computing. Once the domain select is exited or a domain member function returns, execution again becomes sequential.

Access to the Virtual Computer

Virtual processors can be accessed with the array syntax of C from either parallel or sequential code (see figure 4 lines 12 and 23). It is through these kinds of expressions that implicit communication is performed in Dataparallel C. If messages must be passed between physical processors to accomplish a reference, the programer need only be concerned with defining the index of a virtual processor and then referencing a member of the domain structure. No explicit message passing need be

```
1   float   pi = 3.14159; /* Mono-space definitions */
2   float   sum;
3   float   distance;

4   [domain Circles].{/* Domain select statement */
5       float   distance00; /* Poly-space definitions */
6       int     to_far;
7       int     row;
8       int     column;

9       row = (this - &circles[0][0]) / NUM_COLS;
10      column = (this - &circles[0][0]) % NUM_COLS;
11      distance = compute_distance(x, y); /* Member assignment */
12      distance00 = circles[0][0].distance; /* Parallel Reference */
13      if (distance > distance00) {
14          to_far = 1; /* Active/inactive virtual processor */
15      }
16      else {
17          to_far = 0; /* Active/inactive virtual processor */
18      }
19      circumference = pi * radius * 2.0; /* Mono-space read */
20      sum = sum + distance; /* Mono-space assignment */
21  }
22  compute_area(); /* Member function call */
23  distance = circles[0][0].distance; /* Sequential Reference */
```

Figure 4: Some sample Dataparallel C code.

performed by the programer. In fact, Dataparallel C does not provide constructs to a programmer for explicit message passing, all message passing is implicit. Dataparallel C automatically handles any communication of values when references are made to virtual processors which may have been mapped to a different physical processor. The Dataparallel C runtime system is able to recognize and take advantage of references to virtual processors mapped to the same physical processor.

Each individual virtual processor may reference itself by using the reserved word `this`. For each virtual processor the value of `this` is a pointer to its own domain instance. Using the `this` pointer, any virtual processor can locate its position in the virtual machine. In figure 4, lines 9 and 10 show how a virtual processor can compute its row and column number in the array of virtual processors.

Many references to virtual processors are likely to be to neighboring virtual processors, so Dataparallel C provides some special convenience functions to access these neighbors when the dimension of the virtual machine is of low degree, *e.g.* `successor()` and `predecessor()` for one dimensional virtual machine topologies.

Mono Variables and Poly Variables

Dataparallel C defines two new classes of variables: *mono* variables, and *poly* variables.

- Mono variables are, semantically, stored on the front-end processor of the virtual machine. They are defined outside of any domain select statement (see figure 4, lines 1–3). Mono variables are said to exist in *mono-space*, also called sequential space.
- Poly variables are all the variables which exist in the virtual processors. They consist of the domain member instance variables (figure 2) and any additional variables defined within a domain select statement (figure 4, lines 5–8), or from within parallel code. Poly variables exist in *poly-space*, also called parallel space. Poly-space variables define the state of an individual virtual processor. Their contents are unique to the virtual processor on which they are defined. The extent of poly variables is the lifetime of the virtual processor, for domain members, or the activation lifetime for virtual processor automatic variables.

The reserved words `mono` and `poly` can be used as storage class modifiers (in the same fashion as `unsigned` and `volatile`) when the context of the variable is not clear or the programmer wishes to disambiguate a section of code. Dataparallel C does not provide additional storage class modifiers such as *read-only*, *write-once*, or *accumulator*.

Mono variables can be accessed from either sequential or parallel code. Assignment to mono variables may result in communication between physical processors. References to mono variables from within *known* parallel code (textually within the domain select or within a member function) will cause a *broadcast* of the new value to all workstations. References from within routines which may be called from either parallel or sequential code are assumed to not cause side effects, and do not require a broadcast of the new value (see sections 3.3 and 4.2).

Not all poly-space variables are accessible from both parallel code and sequential code. The contents of a virtual processor member may be referenced from either parallel or sequential code (figure 4 lines 12 and 23). However, automatic poly variables that are defined within a domain select (figure 4 lines 5–8) can be referenced only by that single virtual processor.

The Semantics of Synchronous Execution

Synchronous execution of each of the virtual processors is achieved by use of a *Master Program Counter* (MPC). Each virtual processor has a local program counter which is either executing in step with the MPC, *active*, or is waiting for the MPC to catch it, *inactive*. Control expressions such as `if-then-else` break the active set of virtual processors into two groups. Upon evaluation of the conditional expression, one group is active and synchronized with the MPC executing the `then` clause. The other group of virtual processors becomes inactive and waits for the MPC to begin the `else` clause. When virtual processors active with the `then` clause complete, the MPC will begin execution of the `else` clause. During execution of the `else` clause, those virtual processors active during the `then` clause become inactive, while those inactive during the `then` clause will become active. Upon completion of the `else` clause all virtual processors become synchronized with the MPC.

It is important to note that the synchronous execution described above is one of semantics only. Dataparallel C does not require that all virtual processors to actually proceed in lock step, which would require synchronization of all virtual processors at the operator level. Dataparallel C performs actual synchronization only when data dependences exist between virtual processors. The dependencies are described by expressions which reference virtual processors in the active back-end virtual computer. These references can be detected by the Dataparallel C compiler and grouped to minimize the actual synchronizations.

3.3 The Dataparallel C Network Implementations

Implementations of Dataparallel C are available to support a variety of computing hardware: the Sequent shared memory multiprocessors[8], the nCUBE 3200 Hypercube[9], the Intel iPSC/2 Hypercube[10], and networks of UNIX workstations [Nedeljkovic and Quinn, 1992] such as the Sun[11] SPARCstation[12], the IBM RS/6000 series[13], and the Hewlett-Packard 9000/700 series.[14] Our work was done using a network of Sun Workstations.

The network version of Dataparallel C allows the virtual computer to be composed of machines from several vendors and several performance classes. Its ability to utilize a network of computers as a single more powerful parallel computer increases its appeal greatly. It may provide a cost effective way to have both distributed medium performance workstations and high-performance multi-processor computing. We expect the ability to utilize networks of workstations for high-performance parallel computing to become an increasingly important aspect of research in parallel programming environments.

Load Balancing

When running a Dataparallel C program on either a multi-processor, *e.g.* Sequent Symmetry, or a multi-computer, *e.g.* iPSC/2, all processors are usually of the same performance class. This is not likely to be case when using workstations connected to a local area network (LAN). Using workstations of differing performance classes would cause problems if the number of virtual processors allocated to slow machines was the same as the number allocated to faster machines. The faster machines would complete work well in advance of the slower machines. The execution would be limited by the throughput of the slowest machine on the network, resulting in a poor utilization of computing resources. Addressing this problem, the network implementation of Dataparallel C requires a programmer to assign of relative speed to each of the workstations to be used by a Dataparallel C program. Slow machines are given low numbers and faster machines are given higher numbers. Dataparallel C will use the relative performance numbers to distribute the virtual processors among the workstations [Nedeljkovic and Quinn, 1992]. Faster machines will be allocated

[8]Sequent and Sequent Symmetry are trademarks of Sequent Computer Systems, Inc.

[9]nCUBE is a trademark of nCUBE Corporation.

[10]Intel and iPSC are is a trademark of Intel Corporation.

[11]Sun, Sun Microsystems, Sun Workstation, IPX, NFS, and SunOS are trademarks of Sun Microsystems, Inc.

[12]SPARC and SPARCstation are trademarks of SPARC International, Inc.

[13]IBM and RS/6000 are trademarks of International Business Machines.

[14]Hewlett-Packard, HP and HP 9000/700 are trademarks of the Hewlett-Packard Company.

more virtual processors, slower machines fewer. This initial non-uniform distribution of virtual processors among the physical processors provides a form of *static* load balancing.

Static load balancing with a non-uniform initial distribution of virtual processors may not be enough to ensure a balanced distribution of work. Faster machines may have lower actual throughput rates due to other executing programs. Slower machines may be more lightly loaded and provide greater actual throughput. The network implementation of Dataparallel C includes a *dynamic* load balancing scheme. This shifts virtual processors from very busy workstations to idle or not so busy workstations during execution of a Dataparallel C program. Using dynamic load balancing, workstations of different performance classes can be effectively used in a single Dataparallel C program to maximize actual throughput. Faster or lightly loaded workstations would be re-allocated more virtual processors as the overall computation proceeded, regardless of the relative speed initially assigned. The re-allocation of virtual processors through the dynamic load balancing is transparent to the Dataparallel C programmer. It is performed by the underlying Dataparallel C routing library during virtual processor to virtual processor communication.

Compiling and Running a Dataparallel C Program

The Dataparallel C compiler (`dpc-unh`) accepts files with a `.dpc` file extension and compiles them producing to standard C. The C source is then compiled with the standard system C compiler and linked with a special Dataparallel C routing library. Compilation to C makes porting Dataparallel C to new systems much easier. It also allows a programmer to use the very best available optimizing C compiler for the final program.

Running a Dataparallel C program requires a host workstation and the `host` program. The `host` program is run from the host workstation and connects to the other workstations to be used by a Dataparallel C program. If five workstations are to be used, the `host` program will connect to five workstation and execute a copy of the Dataparallel C program on each. In some implementations the `host` program acts as the front-end to the virtual machine. On the network implementation of Dataparallel C the `host` program merely acts as an initiator of copies of the Dataparallel C program and the point to which output is sent.

```
terra    jchaney passwd 2.0
ecology  jchaney passwd 4.0
steppe   jchaney passwd 4.5
asia     jchaney passwd 1.0
snow     jchaney passwd 2.0
hydro    jchaney passwd 1.0
```

Figure 5: A sample `.nhosts` file containing entries for 6 machines. Larger numbers represent faster machines.

The network version of Dataparallel C reads a list of machine names from a file in the user's home directory — the `.nhosts` file contains names of machines to which the `host` program can connect (see figure 5). The machines in this file are used in the order in which they appear in the file. If the Dataparallel C program is to use four machines, then they are the first four machines listed in the `.nhosts` file. If connection with any of the machines to be used is not possible, the entire run is aborted.

The Semantics and Reality of Sequential and Parallel Execution

The semantics of sequential and parallel execution are not quite the same as the reality of running a program on several different physical processors. The network implementation of Dataparallel C starts a copy of the program on each workstation. Each copy has all of both the sequential front-end and parallel back-end. During Dataparallel C sequential execution, all workstations are executing the same statements. Only one of the workstations is defined as semantically being active. In reality, the front-end of a Dataparallel C program is executing on each of the workstations, results from all but one are be discarded. It is not defined which of the workstations is the one which is "active" from the programmers view. Since all are executing the same code on the same variables with the same values, it does not matter which one may be active and which ones have results discarded. Input and output must be handled differently.

Input from a device must be handled from within sequential code. However, it is often the case that members of a virtual processor must have a value assigned from input. During input, data are read on one workstation (a single worker actually reads from the input device). The data are then *broadcast* from the single workstation to all copies of the executing program. If the assignment is to a mono-space variable, all workers perform the assignment. If the assignment is for a poly-space variable (*e.g.* a virtual processor domain member), the input must first be assigned to a mono-space variable. The mono-variable is then broadcast to all workers. The worker to which the specific virtual processor has been mapped will perform the assignment. All other workers will ignore the data. An example of this is discussed in section 4.3.

Data output are handled in a similar fashion. The network implementation of Dataparallel C only supports output to `stdout`, so the output device will always be `stdout` of the `host` program. Programs executing output statements in sequential code produce a single line on `stdout`. Programs executing output statements from parallel code produce one line of output for each active virtual processor. If the program is executing in sequential mode and the output is a mono-space variable, all workers execute the output statements, however all but one are ignored. If the the program is executing in sequential mode and the output is poly-space variable (*e.g.* a virtual processor domain member), Dataparallel C first creates a mono-variable, the poly variable is then *broadcast* to all workers. Output is then done as a mono-variable, all workers execute the output statements, and again all but one are ignored. Output from parallel code is different. From within parallel code all virtual processors must be able to generate output. When code within a domain select executes an output statement involving poly-data, all executing workers send messages to a single collection worker. These communications are sent *point-to-point*. The worker collecting the output forwards it to the `host` program. The host program then sends the data to `stdout` of the controlling terminal.

All of the broadcasting or point-to-point communication required for input and output is handled directly by Dataparallel C, from within the routing library. A programmer need only express assignments in source code form. The penalty paid for all the communication for input and output can be quite high. As will be seen, input and output became the bottleneck of performance for the translated program. The trade off was that of simple program expression and a simple set of input and output statements.

4 The Conversion Process

The problems encountered in the conversion process arise from three primary sources: the Dataparallel C language, its current network implementation, and a structural mismatch between the original program and data-parallel programming. The smallest source of problems was the Dataparallel C language itself. The largest source of problems was the current network implementation. The remaining problems arose from a structural mismatch between the flow of control and information in the sequential program and the flow in a data-parallel translation.

4.1 Language Problems

The language related difficulties arose from two basic problems, the size and topology of the virtual machine are compile-time constants and name space collisions between the application and the Dataparallel C. The first problem makes adapting to different data sizes inefficient, and the second problem makes use of third-party libraries awkward to impossible.

Virtual Processor Constants

Dataparallel C works with arrays of virtual processors of fixed dimension and size. The dimension and size of the arrays must be known at *compile* time, thus fixing the size and dimension of the virtual computer at compile time [Hatcher and Quinn, 1991]. This requirement makes it impossible to deal efficiently with data sets of widely varying size with a single binary. The need to know array sizes at compile time is an efficiency requirement for Dataparallel C. It must be known how the virtual processors will be initially distributed to the various physical processors.

This is complicated by the fact that virtual processors are not first class data objects. A pointer to a virtual processor cannot be passed as an argument to a function. So, one could not simply allocate virtual processors on demand or based on the size of the input data. A version of the Dataparallel C MAPSS program would have to be compiled ahead of time to match the size of the input data.

The MAPSS model uses only two differently sized input sets extensively, the USA and the global data. Others are restricted data sets used for testing new portions of MAPSS code. The restricted data sets are small enough that running MAPSS on a uniprocessor is within reason. Though, additional data sets for use with MAPSS are expected, the number would not cause a significant problem. The requirement of having multiple copies of the same program to utilize differently sized data sets would not present a significant impediment in the MAPSS research.

In the same fashion as the size and dimension of the array of virtual processors, the number of workstations making up the physical hardware must be known when compiling a Dataparallel C program. The reason is also efficiency of the distribution of data and work to the physical processors. Our installation makes 24 workstations available to Dataparallel C programs. Were this number to change, either by additions to the network or by request of workstation owners, the program would have to be recompiled. This restriction does not present significant problems to using the ported Dataparallel C program. Compiling on demand a version of the Dataparallel C MAPSS program based upon the size of the input data and the number of available workstations would not become a burden. It took only 60 to 70 seconds to compile the translated program.

Name-Space Collisions

Dataparallel C reserves `north()`, `northeast()`, `northwest()`, *etc.* as special virtual processor neighbor reference functions, like `successor()` and `predecessor()`. This choice of words is awkward when working within a geographic information system, where these terms are used extensively in the application domain to represent actual compass directions.

We attempted to overcome this problem by textually replacing all uses of `north` in calls to GIS routines with another name, `GIS_north`. References to a GIS structure member called `north` was textually be replaced with `GIS_north`. This requires another pass of the source though a utility such as `sed` and forces the use of artificial names which do not conform the the documented GIS member or function names. This proved very clumsy, and though this was actually tried for short time, it was quickly abandoned. The probability of a debugging nightmare was far too great. For this and other reasons detailed within section 4.2 we were ultimately forced to abandon the use of the GRASS libraries when building Dataparallel C MAPSS.

4.2 Implementation Problems

The implementation of Dataparallel C we used in our efforts is a research project, not a production level product. Its restrictions have a significant impact. Most notable, it prevented the use of third party libraries. Other problems were: lack of compatibility with standard C, the Dataparallel C preprocessor, limited input and output functions, restrictions of the use of `typedef` and `enum` statements.

The Source Tree

Initially we wanted to maintain the MAPSS source tree as both Dataparallel C and standard C, using a generous number of `#ifdef` blocks. This would have allowed continued use of `lint` and `gcc` for syntax and parameter agreement checks in the C source. We knew that much of the C source for the MAPSS program would remain unchanged through the translation.

There were two reasons we were forced abandon C compatibility. First, the modifications required to translate the source were sufficiently extensive to exclude easy standard C compatibility. Now that some initial work has been done, it is possible that subsequent translations can be done while maintaining standard C compatibly. More importantly, MAPSS is itself undergoing continued development. The translation into Dataparallel C would slow and complicate further enhancements to MAPSS were a single source tree to be used.

A Single Source File

The first hurdle we encountered in using Dataparallel C is the execution of the shell script (historically still named `cstar`). It is designed to work with programs composed of a single source file. This does not present a problem when dealing with rather small examples used in research projects. The MAPSS model consists of 21 C program source files and an additional 13 header files. We found a simple workaround — rather than rewrite the shell script, we created a new file which did a `#include` on each of the C source files. The Dataparallel C compiler then had but a single file to compile.

Dataparallel C as a Preprocessor

We encountered many problems when the Dataparallel C compiler was used to parse the source code directly. These problems included reporting syntax errors on lines which contained only comments and incorrectly expanding preprocessor directives and macros. We adopted a solution which used Sun's C compiler to preprocess the source first. We could then use the Dataparallel C compiler on the output from the preprocessor. The Dataparallel C compiler then received only already preprocessed source.

The MAPSS Dataparallel C source was processed using the `-P` option of Sun's `cc` [SunOS]. All the included files, macros, and defined constants would be correctly expanded and comments in the result source were retained. A single large Dataparallel C source file containing the entire source code for the translated MAPSS program could then be used as input to the Dataparallel C compiler.

We did encounter slight problems when syntax errors reported by the Dataparallel C compiler about the MAPSS source had to be traced back to the correct individual C source and header files.

Mono Variables

Guaranteed sequential semantics would imply that modification of a mono-variable (a variable shared among many workers) would require communication between workers. Mono-variables can be placed in such a way that their value is unique to a worker — modification while in parallel code does not result in communication between workers. If a mono-variable is modified from within a routine which can be called from either sequential or parallel code, communication will not occur. Recognization of the modification of a mono-variable from a routine which can be called from either parallel of sequential code would require extensive inter-procedural analysis. The current implementation of Dataparallel C does not perform inter-procedural analysis of modification of mono-variables. Ordinary routines in a Dataparallel C program can be called to execute either sequential or parallel code. An ordinary routine is any routine that is not a member function of a domain declaration. Ordinary routines are assumed to be *pure*, do not contain side effects.

This causes an odd distinction between the two uses of mono-variables. If a mono-variable is modified from within the text of a domain select or from within the text of a member function, communication between all workers will result. However, if a mono-variable is modified from within a routine that is called from within a domain select or from within a member function, no communication will result. We consider this a weak point of the current Dataparallel C compiler.

It was our good fortune to not be bitten by the two meanings of mono-variables. Very late during the tests we noticed a possible flaw in our translation which would have invalidated all the results. A global variable which was being modified by each cell had escaped our search. After detailed analysis we found that this feature of Dataparallel C had cause our translation to not be wrong.

Limited Input and Output Functions

The Dataparallel C compiler we used supports only `dpc_fread()` for input and `printf()` for output. This makes it impossible to rely upon third party libraries for any form of complex input or output. IPW and GRASS rely on additional systems calls to perform file input, output, and error notification.

MAPSS reads over 250 calibration constants from two configuration files, `parameters` and `site`, during initialization. Due to the very limited support for file input provided in Dataparallel C, these were all moved into the code and done as direct assignments to *write-once* mono-variables.

As a final program, having the calibration constants hard coded within the source would place severe limitations on its flexibility. The entire program would have to be rebuilt in order to test a change in any of the calibration constants. Slight changes to the calibration constants are frequent in the MAPSS research. Hard coding also makes it impossible to use a single binary with multiple calibrations. The calibration constants used with various data combinations vary in several places. The calibration for the global data differs from the calibration of the data for the USA.

Though this may sound very bad, and the permutations awful, we had to recognize again that demand compilation would take only about one minute. Developing a utility which would translate the contents of a parameters file into a Dataparallel C source file containing assignment statements could be easily done.

New Types and Enumerations

The Dataparallel C compiler we used does not accept enumerations or newly defined types, i.e. `enum` and `typedef`. The MAPSS C code began with a large number of both `typedef` and `enum` statements. Many new types are used in both the IPW and GRASS third party libraries. GRASS makes use of many enumerations. Four methods were considered to over come these problems.

1. Rework the IPW and GRASS libraries.
2. Alter the text output from the preprocessor using UNIX utilities such as `sed` and recompile IPW and GRASS.
3. Modify the Dataparallel C compiler to correctly deal with `typedef` and `enum` statements.
4. Abandon the use third party libraries.

Option 1 would require a tremendous amount of time and verification of the altered libraries. It would mean that a special version of both libraries would have to be maintained so long as the Dataparallel C MAPSS program was useful. Both IPW and GRASS are administered by groups outside of the researchers associated with MAPSS. The expected effort was far too great to attempt this workaround.

Option 2 would require named structures be used in place of new type references and enumeration values replace the enumeration identifiers for the source of IPW, GRASS, and MAPSS. This was abandoned for the same reasons was option 1.

We never considered making changes to the Dataparallel C compiler such that it could accept a broader range of input and output functions, option 3. We did not feel it was reasonable to expect a typical user to make changes to the compiler to suit the program. Since our purpose was to test the translation of an existing application, we considered it inappropriate to make extensive changes to Dataparallel C compiler itself.

Use of option 4 would require input and output be done using plain ASCII files and the Dataparallel C input and output functions.

Regardless of what was done in the IPW and GRASS libraries, all uses of `typedef` and `enum` in the MAPSS code would have to be changed. Each `typedef` was changed into a named `struct` and the enumerations were changed into `#define` constants.

No Third Party Libraries

For the reasons stated in sections 4.1, 4.2, and 4.2 use of IPW for input and GRASS for output became impossible. Plain ASCII files were created which corresponded to the IPW input and an ASCII file was created as output from Dataparallel C MAPSS.

There were two side effects to reading and writing ASCII data. Normally MAPSS calculates PET internally, however, there are provisions for reading PET values directly from input files. Reading PET as an input would decrease the number of inputs significantly. Instead of reading 12 monthly values for temperature, precipitation, vapor pressure, wind, and single values for elevation and mask, the ASCII version of the input would contain only 12 monthly values for temperature, precipitation, PET, and a single value for the mask. The over all reduction was from reading 50 input values per cell to 37.

The second side effect of reading and writing ASCII data was the increase in the size of the input and output files. Instead of reading 10 mega-byes of binary data, over 36 mega-bytes were read. Instead of writing 14 mega-bytes of data over 30 were created. The increased size of the input is in spite of reading the PET as an input. As will be seen, the amount of time spent in input and output became a dominating portion of the total run time of the Dataparallel C program.

No Dynamic Load Balancing

The non-uniform number of iterations required for each cell has a significant implication for dynamic load balancing. Were one worker to have a large number of computationally "cheap" cells, it could complete it computation very quickly. Other workers may have very few cheap cells and would require a much longer time to complete. In addition, a worker that is under a heavier computation load (from the execution of unrelated programs), or has a slower processor, would take longer to complete than a lightly loaded, or faster, processor. The network implementation of Dataparallel C balances computational loads by dynamically reallocating virtual processors among physical processors *whenever processors communicate*. Unfortunately, no significant communication occurs between cells in our application, and hence the computational load is never dynamically balanced.

4.3 Structure Mismatch Problems

The input and output for MAPSS represents a two dimensional map. The borders of the map are defined by a mask, which is either zero or one. When the mask is a zero (outside of the mask), calculations for vegetation are not performed. Data values for cells outside the mask are output, but they are used only for filling in the entire map. The initial translation of MAPSS into a Dataparallel C program was done by creating a two dimensional array of virtual processors (see figure 6). Virtual processors for cells outside the mask would be idle, an effective *no-op*. Since as much as 76 percent of the input data represented cells outside the mask, a very large number of virtual processors would remain idle throughout the entire computation.

Later, as testing progressed, another configuration of the Dataparallel C virtual computer was used. Instead of the two dimensional array of virtual processors, a one dimensional array was used (see section 5.1). All of the cells which were outside of the mask were removed from the input data. This was a change to the data only, no additional changes to the translated program were made.

```
#define YEAR 12
domain Cell {
    int     mask;
    int     veg_class;
    float   tmp[YEAR];
    float   ppt[YEAR];
    float   pet[YEAR];
    ⋮       /* Other Domain Members */
};

domain Cell cell[ROWS][COLUMNS];
```

Figure 6: A sample of the Dataparallel C domain declaration for MAPSS.

Input and Output Conversion

The first functional change to the MAPSS program was how and when input and output are handled. The original program would read a cell, process the cell, and output the results. One cell at a time was processed in a *read-one, compute-one, print-one* loop. The sequential program was not structured such that input and output sections were clearly defined within the code as separate from the compute section. Specifically, a cell which was masked out would be recognized and output directly from the input routines. Cells which lacked sufficient precipitation would be output from the compute routines early. The Dataparallel C program would have to be different — it would have to read all of the input data, process all of the input data, and then output all of the results, *read-all, compute-all, print-all.* The transformation was accomplished by partitioning the program into three distinct sections. All input functions were moved to the beginning section. This was be followed by single domain select, which performed the cell calculations on a single back-end SIMD virtual computer. Finally, all the output functions were pushed to the final section of the program. Both the input and the output sections of code were executed from within in sequential code.

An example of how the input was done in the translated program is shown in figure 7. A consequence of the SIMD nature of Dataparallel C is that line 9 would be executed on one worker. It would then broadcast the value of `buffer` to all other workers. All workers would execute the statements on lines 13 and 14. The worker to which virtual processor `cell[row][column]` had been mapped would perform the assignments on line 15 and lines 18–20.

```
int         mo;
int         row;
int         column;
int         mask;
char        buffer[BUFSIZ];
char        *ch;

for (row = 0; row < ROWS; row++) {
    for (column = 0; column < COLUMNS; column++) {
        if (!dpc_fread(buffer, sizeof(buffer), 1, in_file)) {
            (void) printf("Read failed, exiting");
            exit(MALFORMED_INPUT);
        }
        mask = atoi(buffer);
        ch = &buffer[MASK_OFFSET];
        cell[row][col].mask = mask;
        for (mo = JAN; mo <= DEC; mo++, ch += NUM_OFFSET) {
            (void) sscanf(ch, "%f%f%f", &ppt, &pet, &tmp);
            cell[row][col].ppt[mo] = ppt;
            cell[row][col].pet[mo] = pet;
            cell[row][col].tmp[mo] = tmp;
        }
    }
}
```

Figure 7: A sample of Dataparallel C input code used in MAPSS.

Data output are handled in a reverse fashion, values are broadcast to all workers, and a single process which actually does the write to the output device. Figure 8 shows the Dataparallel C translation of MAPSS output code. Lines 5 and 6 result in a broadcast of the the member `veg_class` of virtual processor `cell[row][column]`.

Further explanation of both input and output for the network implementation of Dataparallel C is given in section 3.3.

An additional consequence of the transformation from the original *read-one, compute-one, print-one* to the data-parallel *read-all, compute-all, print-all* was a dramatic increase in the amount of memory required by the executing program. The sequential program fit within a constant amount of memory, about 1.3 mega-bytes. The translated program would require memory as a function of the size of the input; the actual memory required would be the sum of the memory used by each workstation. The domain definition we used required nearly 2000 bytes of storage for each virtual processor. The data-space requirements of the USA data set in with two dimensional array of virtual processors would require over 270 mega-bytes of memory. The global data set would require over 500 mega-bytes of memory. Using 10 workstations, each image would be in excess of 27 or 50 mega-bytes. We could have have found a more space efficient representation for the domain definition, but we found our testing bound by total compute time, not memory restricted.

```
int         row;
int         column;

for (row = 0; row < ROWS; row++) {
    for (column = 0; column < COLUMNS; column++) {
        (void) printf("r%d c%d:  %d",
            row, col, cell[row][column].veg_class);
    }
}
```

Figure 8: Some sample Dataparallel C output code.

Global Variables

Through no fault of Dataparallel C, it is difficult to translate C programs which rely upon a large number of global variables. The global variables used in MAPSS fall into three classes, those which are independent of the cell calculation, those whose value changes during a cell calculation, and those whose value changes during execution but does not affect cell calculation. Those which are independent of the cell calculation can be treated as *write-once* variables and can be left as global variables. Their values can be broadcast to all workers without concern of modification by cell calculation. Hidden among the *write-once* variables were the second class global variables, those whose values changed during computation. These variables obscure a lot of inter-procedural information. The final class are the *accumulators*; their value may change, but the new value does not affect following cells and does not invalidate previous calculations. Modification of an accumulator does require communication, but it is infrequent and can be overlapped with other useful computation.

The over 250 calibration constants represent global variables which are fixed during an initialization phase and need not be copied into the virtual processor instance record. These are declared as `mono` variables in the front-end. All copies of the program perform the assignments of values to these variables. They are used exclusively as *rvalues* after initialization. No communication need be done prior to a reference to a calibration constant.

The second class of global variables, those whose values were very important to individual cell calculation, were either folded into the domain declaration, defined as automatic poly-variables within the domain select, or placed such that their modification did not affect other cells by changing a threshold value or causing needless and expensive communication overhead.

The final class of global variables, the accumulators, create implicit dependences between virtual processors. Though none of the cells in the MAPSS program have any explicit dependencies on any other cell, there is a value which is accumulated for an entire run. The global variable, `total_final_exceed` is conditionally incremented in the code, from within the text of the domain select. Changing the value of a mono-variable from within the text of the domain select will result in communication between the physical processors. The effect an accumulator can have on the performance of a Dataparallel C program depends on the location of the reference. It was modified only at the end of the domain select block. This location meant that all cell calculation could proceed without need for communication between cells.

SIMD Model of Computation

Upon first analysis MAPSS seemed perfect to run in a SIMD environment. The SIMD model of computation carries with it the idea that all statements are executing in lock step. The MAPSS model does not fit well into that concept. Each data cell reached equilibrium independent of all others, however some cells require only a few iterations to find equilibrium, while others require several thousand iterations; masked cells actually require zero iterations. This carries two important implications about MAPSS under a SIMD model of computation. First, the amount of *time* it takes each cell to achieve equilibrium will vary greatly from cell to cell. Second, the actual function active on different cells will vary instantaneously. The active function on cell $c_{x_1 y_1}$ may be distributing available water, the active function for cell $c_{x_2 y_2}$ may be changing the grass LAI, while the active function on cell $c_{x_3 y_3}$ may have reached equilibrium. If the entire program were forced to block while each virtual processor found its way to equilibrium the compute time would be far longer than that found on a uniprocessor. Only at the highest level is the MAPSS program SIMD. The same program is applied to each cell. Upon that application, the cells march at differing steps.

Fortunately, the Dataparallel C does not require strict SIMD execution, but instead only enforces synchronization as *necessary*. So, while our application is inappropriate for strict SIMD execution, it is appropriate for MIMD implementations of data-parallel programming. Dataparallel C is a MIMD implementation of data-parallel programming

4.4 Overall Changes

We attempted the simplest translation possible. We created no member functions, we left unmodified any code which did not require modification. The actual number of changes to the MAPSS functions for the translation to a Dataparallel C program were small. Most had to do with argument passing from the virtual processor instance record, structure dereferencing, input and output, and creation of the virtual processor instance record. The majority of the changes were confined to a small number of files and functions. Nearly all of the time required for translation was used finding shortcomings with the network implementation of the Dataparallel C compiler, specifically, using it as a preprocessor.

5 The Timing Results

We performed our timing tests in an environment which we believe will be characteristic of the use of production network distributed programming systems.

5.1 Timing Environment

The environment for the timing tests consisted of a network of Sun Workstations connected by a LAN. All workstations used as workers for the timing tests were of the same performance class. Each was either a Sun SPARCstation 2 or Sun SPARCstation IPX. The workstation running as the `host` program was a Sun SPARCstation IPX. The workstation running the `host` program was not used as a worker. The number of workers varied from two to 11. More workstations were available for use as workers, but they were of a lower or higher performance class. As of this writing, the single workstation configuration of Dataparallel C for Sun workstations has not been implemented.

No attempt was made to gain exclusive access to any workstation in the network. This was done for two reasons. First, we believe that this represents how network computing will most likely be used. This complicates determination of optimal runtime of the translated program. For example, it is known that file system backups were performed on the machines being used during the timing tests. Secondly, the workstations are actually owned by several research groups. We would not have been allowed exclusive access to someone else's machines.

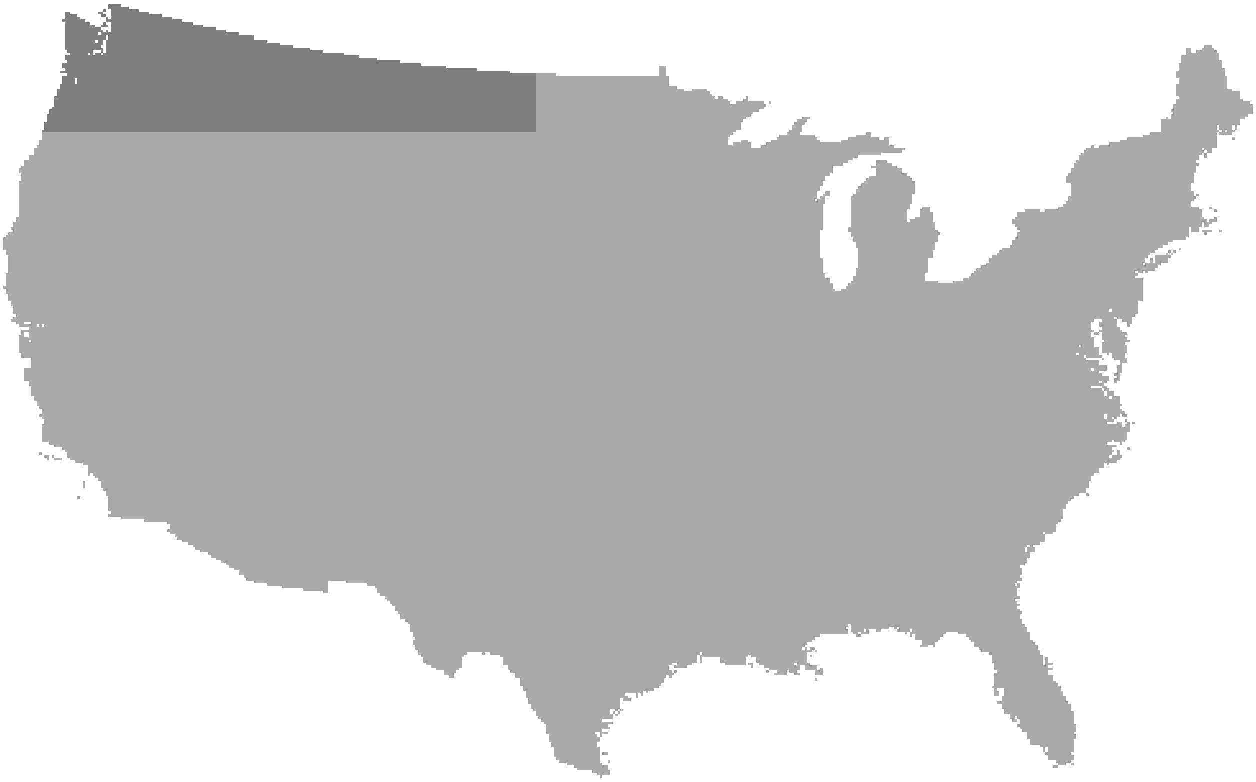

Figure 9: The large data set is shown in light. The medium data set is shown in dark.

Input Data

Four test files were created for testing the Dataparallel C translation of MAPSS and making control runs of the sequential program: tiny, small, medium, and large. The tiny file had only six data cells and was used only for testing correctness and termination of the translated program. The small data set contained 11 rows and 51 columns, 561 cells. We used this data set when we performed tests to determine how long it took the `host` program to connect to each workstation and start a copy of the Dataparallel C program on the workstation. The medium data file contained 10,251 cells of data, 51 rows by 201 columns, see figure 9. Of these, 45 percent were outside the mask. This was the data used to obtain the timing results, a file of approximately 2.7 mega-bytes. The large data file contained the all the data for the USA, 138,650 cells, a file of approximately 36.6 mega-bytes. Originally this was to be the data file used for the timing tests. However, the amount of time required for a single run was too great to make data collection easy — it was considered unacceptable to allow the program to run on another's workstation for such a long time. The sequential program required more than 24 hours to complete a single run with the large data set. In addition, the memory resources required by the large data set was more than a two workstation Dataparallel C program could allocate.

The data files, both input and output, were kept on a SCSI disk physically mounted on the workstation from which the `host` program was run. The data disk was NFS mounted on all other workstations.

The Impact of No Dynamic Load Balancing

For this exercise dynamic load balancing available in the network implementation of Dataparallel C proved of no use. There were three reasons for this:

1. the computation for each cell in MAPSS is independent of all other cells,
2. the location of the global accumulator was such that communication, *i.e.* dynamic load balancing, would take place only after all compute intensive portions of MAPSS was complete (see F.1, lines 159–160 and F.2 lines 498–543), and
3. when we actually tried to use dynamic load balancing, the Dataparallel C program would never terminate — it would hang.

Therefore, no effective dynamic load balancing took place during a run of the Dataparallel C MAPSS program. This is most unfortunate. It would have been very interesting to see how dynamic load balancing would effect the average run times when workstations of differing performance classes were mixed while running the translated program. In addition, dynamic load balancing would mitigate some timing anomalies created by workstations being used by other programs during timing runs. Finally, it would have been useful to see how a Dataparallel C program would perform with more workstations, as many as 24 could have been employed.

Pruning Masked Data

Because so much of the data in the input files represented cells which were outside the mask (45 percent for the test data file, and 45 percent for the lower 48 states and 76 for the globe), a very large number of the virtual processors were effectively idle throughout the entire computation. An uneven load distribution may result as each virtual processor outside the mask would behave as a *no-op*. One workstation could have nearly all it virtual processors a masked out, allowing it to complete quickly. Another workstation could have no masked cells at all, requiring it to work much longer. This made effective load distribution much more difficult, especially if the only the static distribution could be used. We believed that removal of the masked data would result in a significant reduction in overall run time by making better use of the computing resources. We made a second copy of each of the four data sets and *pruned* all of the masked data cells from the copy.

Removal of the masked data points did make a significant reduction in the size of the virtual machine. The test data set would now contain only 5,623 data cells, down from 10,251. The large data set for the USA would decrease from almost 140,000 to fewer the 78,000. The effect on the world data set would be most dramatic, the number of cells would drop from over 250,000 to fewer that 63,000. It is very important to note that this represents a change in the data only. No changes to the code were required to remove the masked data cells.

5.2 Method of Timing

Several runs for each configuration of both the sequential MAPSS program and the Dataparallel C translation were done to smooth over anomalies in the test results. All tests were begun during off peak utilization times, usually beginning at 22:00 hours and completing by 07:30 hours the following morning.

Elapsed runtime results were gathered using the UNIX utility `/bin/time`. Reported is wall time for the `host` process only. The amount of CPU or system time reported by `/bin/time` is not useful in this situation. Very little of the total amount of CPU or system time is spent on the host processor. The time to connect to each of the workstations used in a run is included in the run times observed using `/bin/time`.

Read, write, and work timing results for the Dataparallel C program were obtained by adding a calls to the UNIX library function `time()` in key sections of the translated source. Calls to `time()` were also added to the `host` program to determine the amount of time required to connect to workstations and start a copy of the Dataparallel C program on each. The `time()` function returns the number of seconds since 00:00:00 GMT, 1 January 1970. The ending time was then subtracted from beginning time to give an elapsed time.

Determination of the amount of time spent in input and output for the sequential program proved more difficult. Input, work, and output are done on a per cell basis, there is no clear separation of any of the three. The sequential program was compiled with profiling enabled, the `-pg` option of Sun's C compiler [SunOS]. It was then run with normal input. The profile output, `gmon.out`, was analyzed using `gprof`. The total time for each of the input and output routines was added (by hand). These numbers were then used to determine the percent of the total time required for input and output. These percentages were then applied to the results of running the non-profiled sequential program.

Averages

A large degree of variability is shown in the timing results. Exclusive access to each of the machines on the network was never attempted, so this is not very surprising. However, the very wide degree of results is rather shocking. It can be partially explained by the lack of dynamic load balancing of the final Dataparallel C program. If dynamic load balancing worked the results may have shown closer correspondence.

Because the standard deviation was large, we calculated a new average by discarding outliers. To do so we first calculated the true average and then threw out run times that were more than two standard deviations away from the true average. We then calculated a new average. This did not change the results significantly. The averages shown in the graphs are the new averages, except where we also have a line for the true average.

Ideal Time

The *ideal time* is calculated dividing the sequential time by the number of workstations. If the program were perfectly parallelizable and perfectly translated, the runtime of the parallel program would equal this ideal time. This calculation of ideal time does not take into account the sequential nature of the input and output sections of the translated program and as such is biased toward the sequential time. A better calculation of ideal time would take into account those portions of the program which are inherently sequential. Often all input and output are considered inherently sequential.

5.3 Dataparallel C MAPSS

Figures 10 and 11 (page 31) summarize the timing results (for both the unpruned and pruned data, respectively) for the Dataparallel C version of MAPSS on the network of workstations with the medium size data set. Results from the individual runs can be found in the appendix, sections D.1, D.2, E.1, and E.2. The ideal time is shown as the lower line in each graph.

It is important to note that the amount of time required for the Dataparallel C program to complete initially shows a nearly linear decrease as the number of workstations is increased. Both sets of results show the graph beginning to flatten, become horizontal, beyond six workstations — the gap between the Dataparallel C program and the ideal time increases as the number of workstations increases beyond six. This does imply that this program under these conditions does not scale well beyond six workstations.

Unpruned Data

The time required for each run of the Dataparallel C version of MAPSS varied by over 8,000 seconds on the three workstation runs. The time required by the sequential program varies by as much as 60 seconds. The average total time required for a two workstation run was over 150 seconds longer than the average sequential time and over 1,800 seconds longer than the ideal time for two workstations. The 11 workstation run averaged over 1,500 seconds longer than the similar ideal time. The best average time was achieved when using 11 workstations, 1,858 seconds. The best overall time was with nine workstations, 1,603 seconds. The use of 11 workstations for the Dataparallel C MAPSS model did not cut in half the amount of time required by the sequential program on a single workstation.

The timing results for the unpruned data show a clear bimodal distribution. We continually found it difficult to obtain consistent timing results while working in a production environment. More consistent results could be obtained from a network of dedicated workstations. However, a goal was to test the performance of the translated program in an environment similar to one in which it would be used. Reporting results of the translated program when run on a dedicated network would be misleading to potential users of network computing.

Pruned Data

Figure 11 shows the results of running the Dataparallel C program in the pruned medium size data set. The runs using the pruned data did uniformly show a best time below the best time for the unpruned runs. The best time for the pruned runs was with 11 workstations, 1,055 seconds.

We found it quite surprising that the average run time when using either the pruned or unpruned data remained basically unchanged for both the sequential MAPSS program and the Dataparallel C MAPSS translation. The same source code was used to generate a program to run with the pruned data. Instead of a two dimensional array of virtual processors, a single dimensional array was used. Fundamentally, the data pruning represents only a change in the amount data input and output. The amount of work done does not appreciably change. We did not expect the time for the sequential program to change a great deal with the pruned data. We did expect to find a significant time reduction with the Dataparallel C translated program when run with the pruned data. We expected the work distribution between physical processors to be better allocated, resulting in higher overall throughput and lower overall run times. We hoped this would make a good comparison against the work distribution for the unpruned data.

We investigated possible causes for the similarity in run time between the pruned and unpruned data a bit further for the Dataparallel C MAPSS program. On each worker Dataparallel C creates `for` loops to emulate to requested number of virtual processors. Each control expression, `if-then-else`, creates a `for` loop that iterates through all of the virtual processors on each worker. Even if there are no masked cells, each worker iterates though all virtual processors checking for masked cells. The body of the `for` loop this is never executed when using the pruned data, however simply iterating though all the virtual processors costs time. It is our hypothesis that iterating through all of the virtual processors on all control expressions caused the pruned data runs to show no significant differences in total runtime. (See appendix F.1, lines 167–188 and F.2, lines 564–613).

How the Time is Used

The results in shown figure 12 and 13 show time used in the Dataparallel C program for input and output. The sequential version of MAPSS, when run with the unpruned data, requires less then 27 seconds (only 0.8 percent) to read data and just over 70 (2.1 percent) seconds of a average total of 3,344 to write the data (see section 5.2 for a description of how these numbers were calculated). The amount of time required for the Dataparallel C MAPSS program to perform input and output increased as the number of workstations increased. Using the unpruned data the Dataparallel C MAPSS program averaged just over 70 seconds when using two workstations and over 165 seconds when using 11 workstations. The read time for the pruned data was 43 seconds for two workstations and 106 seconds for 11 workstations. When using the unpruned data, the time to write data went from 117 seconds when using two workstations to 632 seconds when using 11 workstations. The time for write with the pruned data went from 66 seconds with two workstations to 437 seconds with 11 workstations.

Figures 12 and 13 give a clues as to why the curve in figures 10 and 11 begin to flatten out as the number of workstations increases. The amount of time required for both input and output steadily increase as the number of workstations increases. This came as no great surprise. As the number of workstations increases, the amount of communication required to distribute and collect data also increases. We found surprising just how much faster the write time increases over read time.

The ideal time shown in figures 12 and 13 assumes that parallel input and output can be utilized. The Dataparallel C MAPSS program was not able to perform parallel input or output. Parallel input and output are topics for research in data-parallel programming. Without parallel input and output, the an ideal translated program would have shown a flat line for both input and output across all tests.

Figures 14 and 15 shows how the time required for work decreases linearly as the number of workstations increases. These curves do not flatten as that shown in figures 10 and 11. This is exactly how we would hope the work curve to look. These figures show that the work for the Dataparallel C MAPSS program, under these conditions, does scale well as the number of workstations increases.

Figures 16 and 17 demonstrate how the input and output time begin to dominate the total amount of time required. Examining the unpruned data, with two workstations less than four percent of the total time is spent doing input and output; with 11 workstations over 40 percent of the total time is spent doing input and output. Input and output have become the bottleneck. Parallelizing the work has been of little consequence, much of the time gained by computing the cells in parallel has been lost to reading input data and writing results.

After we had collected timing results for the translated program we collected some statistics on how much time the `host` program was taking to connect to each of the workstations to be used in the virtual computer. Figure 18 shows how the time required to connect and initialize processes on remote workers shows a steady increase as the number of workstations increases. Though the amount of time required to connect to each of the workstations participating in the virtual computer does show a steep increase, it accounts for only a small percent of the total run time, for this number of workstations. Were the number of workstations to increase into the hundreds or even thousands, connect time could become the bottleneck.

6 Conclusions

We learned several lessons from our experience translating and running an existing scientific program. We address these lessons to three groups of people, parallel language designers, parallel language implementors, and applications programmers.

6.1 Parallel Language Designers

Parallel programming languages should permit dynamic data sizes. MAPSS had only two sizes of input data. Having two copies of a program is not a significant problem. However, many other applications will have many differing sizes of input. Requiring that the parallel program be compiled on demand will not solve all cases. Some programs will need to be dynamically sized during execution.

Parallel programing languages should support more than just parallel computing; they should support parallel input and output. The support should be built into the language, not simply tagged on as a library. The MAPSS example proved that reducing the time spent in the compute intensive portions of a program will not always result in a shorter runtime. We expect many applications to have the same needs — support for parallel input and output will become imperative.

Parallel languages should not reserve names which are likely to have a meaning in any potential application domain. The easiest way to avoid these types of conflicts is to use a common prefix for all function names and identifiers.

6.2 Parallel Language Implementors

Parallel language implementations should expect to work with third party libraries. Most of the time spent during the translation of MAPSS was spent overcoming the inability to use IPW and GRASS.

Parallel language implementations and environments should support all input and output that sequential programs are accustomed to using. It may be that additional input and output routines are required to support additional or expanded functionality, *e.g.* parallel input and output.

Production users require production implementation, including error reporting and bug detection. Parallel language environments should provide a `lint` like utility to check program source. This utility should be able to check function argument match in type and number, ensure conformance of assignment statements, and report syntacticly incorrect code in a way to assist an application developer.

Applications measure performance from start to finish. Sequential bottlenecks, such as input and output, can dominate any performance gains from parallelizing inner loops. For compatibility reasons, most programs will need to have available the semantics of sequential execution.

Implementations of parallel languages should have some way to recognize and adapt to the compute environment. Dataparallel C requires that the number of workstations be known at compile time. It would be much more convenient to simply specify the number of workstations at runtime. This may be closely tied to the ability of having dynamic data sizing.

6.3 Application Programmers

Application programmers should not use global variables to hide inter-procedural information except where no alternative exists. Access to global variables should be as limited as possible. If synchronization or sequential semantics is required when modifying a global variable, make it as clean and clear as possible. If variables must be global, try to make them read-only or write-once.

7 Future Work

We believe that many additional experiments could come from the work described here. We would like to attempt to reduce the run time of the Dataparallel C MAPSS model by taking advantage of some information known about the internals of the Dataparallel C routing library. This is special information which is not generally available to an application programmer. However, if we can show that a significant reduction in run time is possible by its exploitation we could justify making that information available to an application programmer, in some form.

Additional experiments with the Dataparallel C MAPSS program by removing the code which deals with masked data cells would be of great use in determining why the runs made with the pruned data showed no significant performance improvement over the non-pruned runs (see section 5.3). We found that simply changing the data did not show the reduction in run time as was expected. We believe that we could show a significant reduction in run time by removal of the code which deals with the masked data cells. This is a specialization of the program beyond our initial goal (it presupposes use of the pruned data), but it could point to some information which would be valuable to language designers and implementors.

Obtaining results from a network of dedicated workstations would be very interesting. All of our timing results were obtained from a network on non-dedicated workstations. Though we tried to minimize the interference from other programs by running the tests during off peak times, at night and on weekends, we could never find an optimal runtime for the translated program. Reporting optimal runtime to scientists as expected runtime would be misleading, but it would be helpful in benchmarking other translations.

Changing the routing library for the network implementation of Dataparallel C so that dynamic load balancing would occur without the need for inter-cell communication would allow some very interesting additional experiments. The performance of the Dataparallel C MAPSS program was hindered by the requirement of communications to force dynamic load balancing.

It would be interesting to find how this same application runs on other implementations of Dataparallel C. Other implementations may not have the same bottleneck on input and output as does the network implementation. Specifically, the shared memory implementation for the Sequent would not have to rely on slow network transition time, it could write data directly into array segments shared among workers.

We would like to investigate how easily MAPSS could be ported to other parallel programming languages. The translation into Dataparallel C could serve as a base case for determining what languages may prove easy or more difficult. How might other parallel languages perform with this same application? Would input and output prove to be the bottleneck for all languages?

Acknowledgements

We would like to express our thanks to some people who assisted us in this work. Phyl Crandal provided us with the Dataparallel C compiler and runtime system. She was always patient when we reported problems — even when the problems were our own, not those of Dataparallel C. Dr. Michael Quinn provided us with a great deal of information about how Dataparallel C handles communication during the modification of mono-variables. He also gave us some invaluable tips on

how we could reduce the time required to output data. We hope to take his advice and perform additional tests for another report. Dr. Ronald Neilson provided us with the MAPSS model and important information about its inner workings. He made sure we addressed the needs of scientists outside of Computer Science.

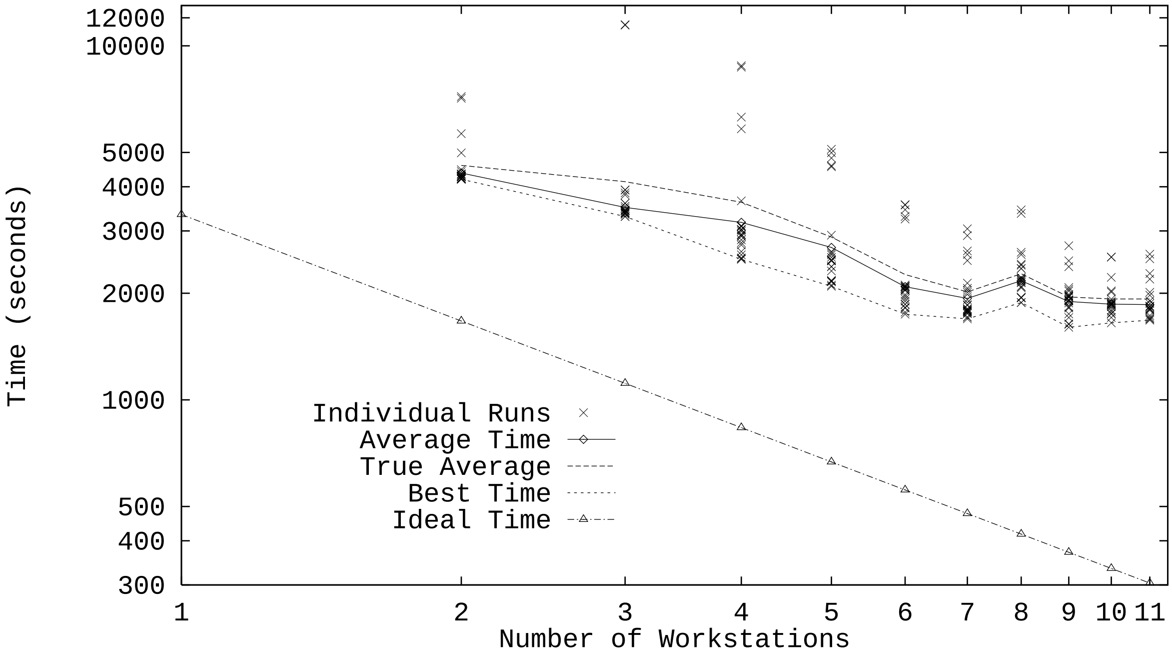

Figure 10: Timing Results (Unpruned).

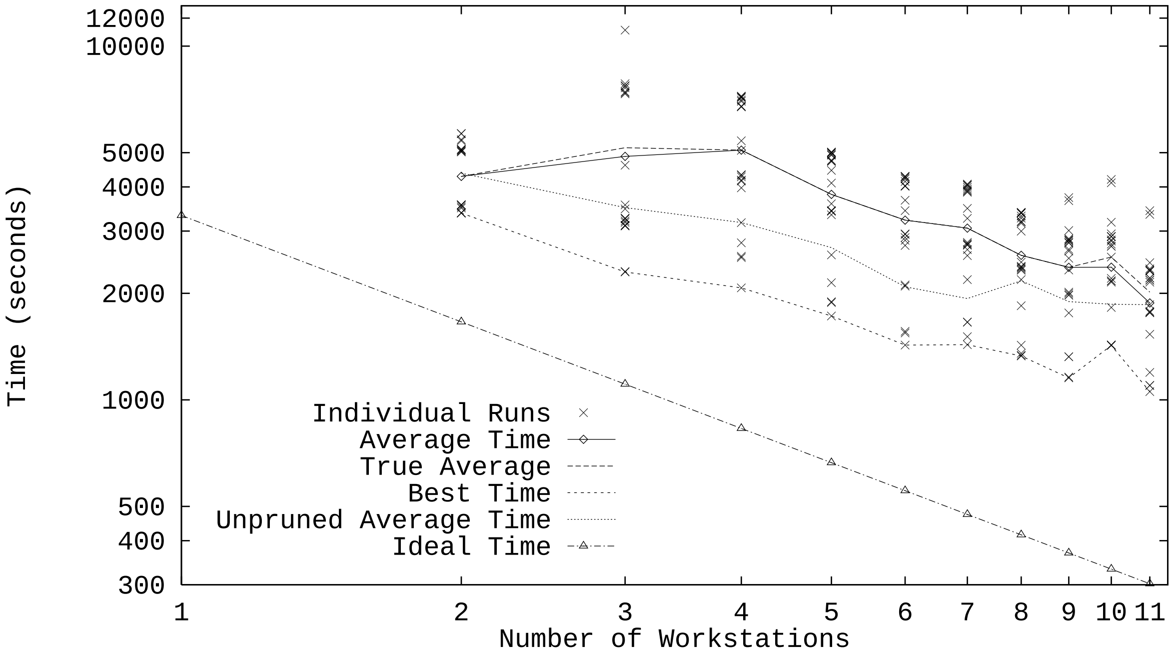

Figure 11: Timing Results (Pruned).

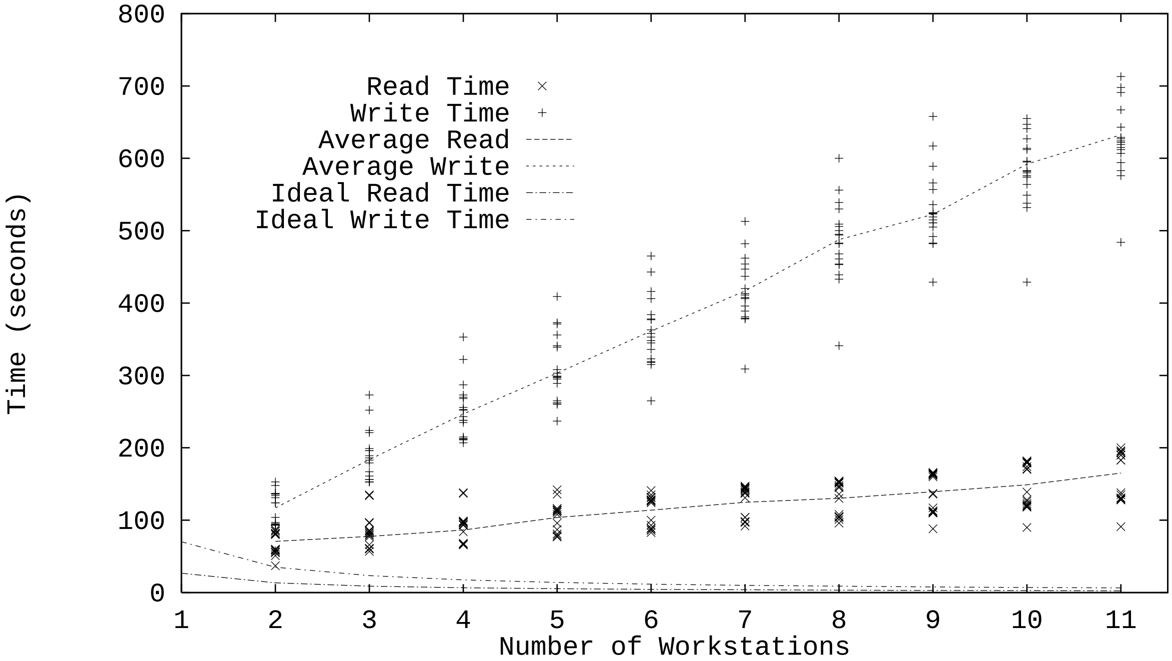

Figure 12: Read and Write Timing Results (Unpruned).

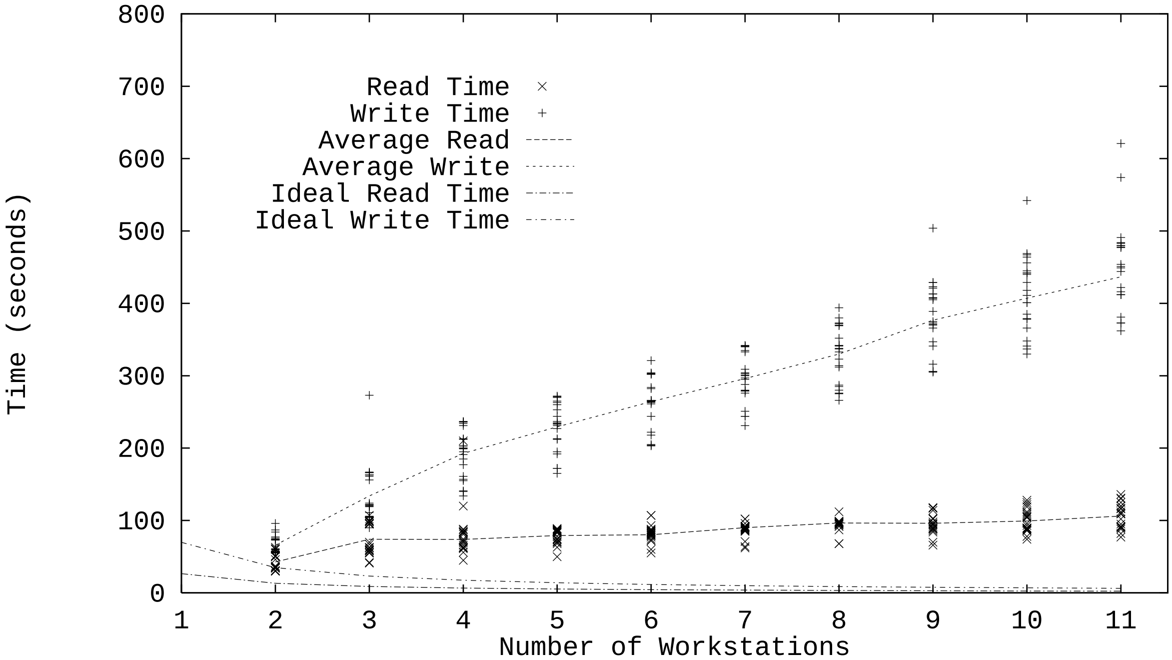

Figure 13: Read and Write Timing Results (Pruned).

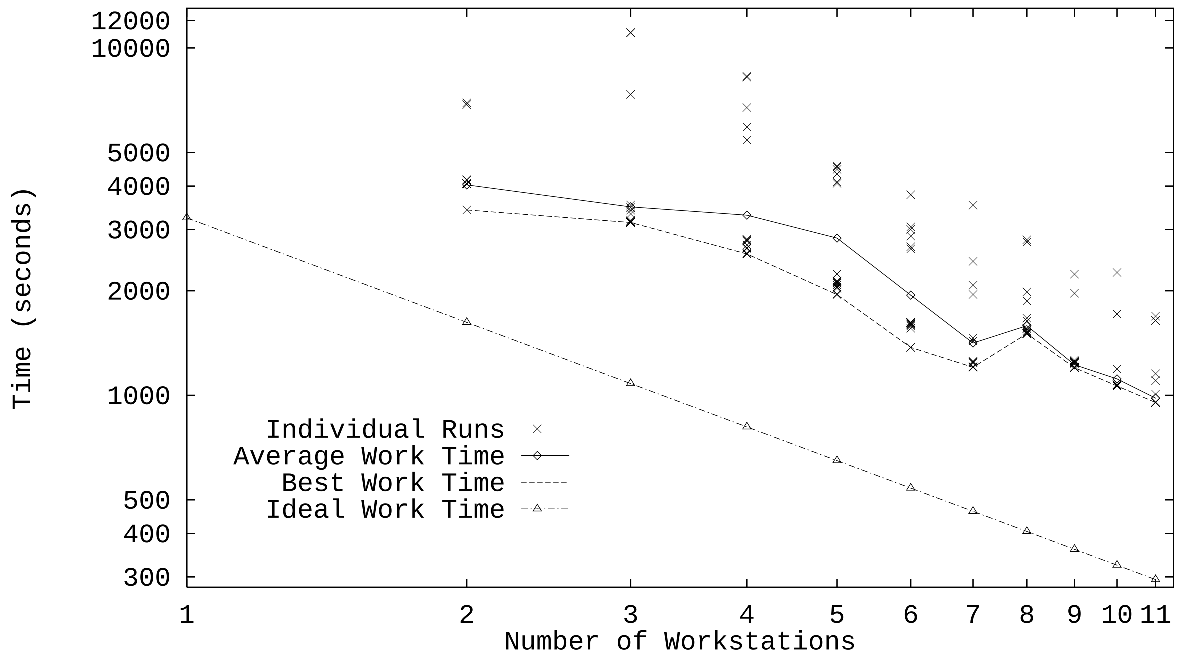

Figure 14: Work Timing Results (Unpruned).

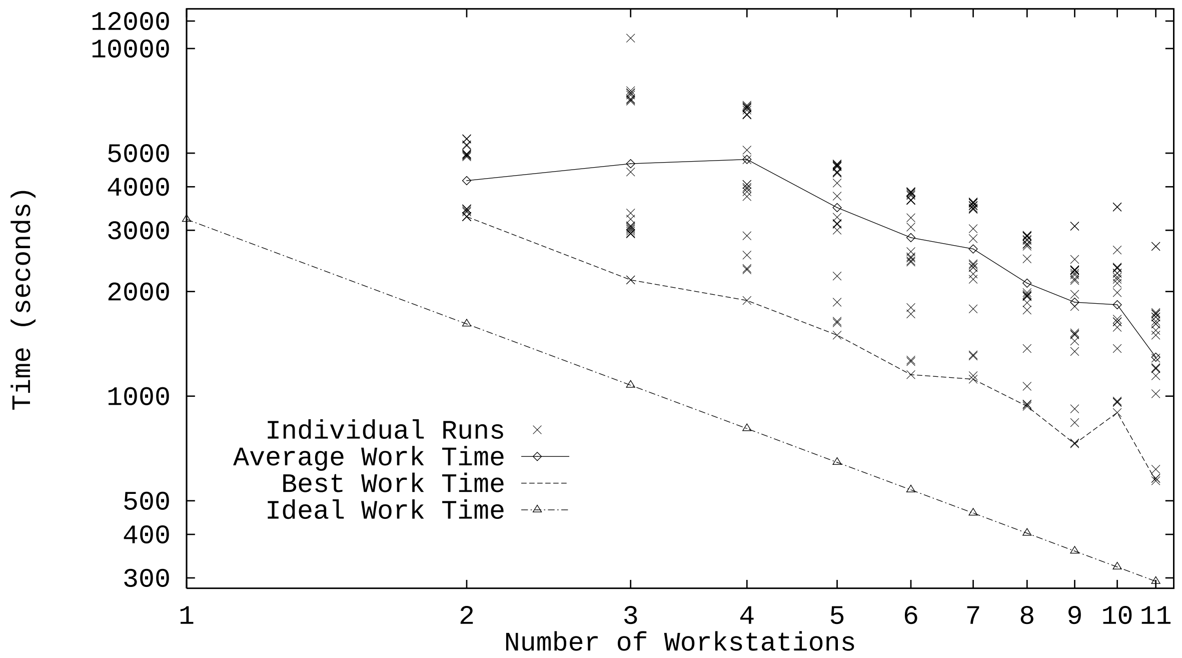

Figure 15: Work Timing Results (Pruned).

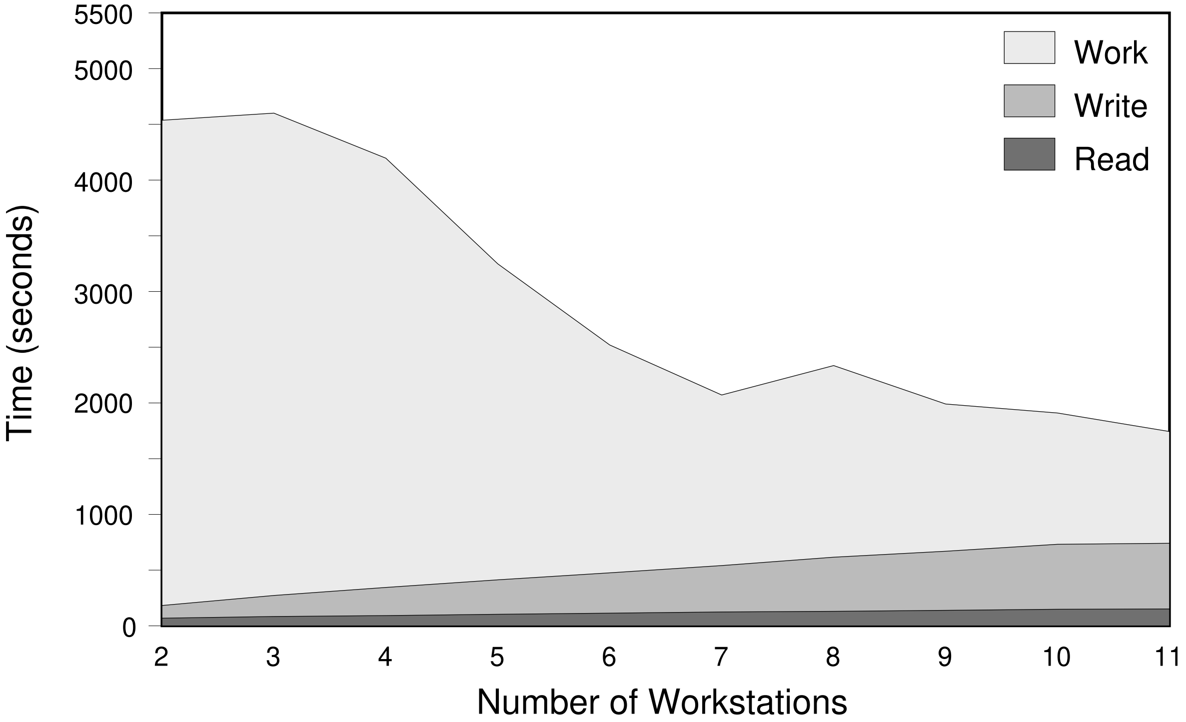

Figure 16: Summary Timing Results (Unpruned).

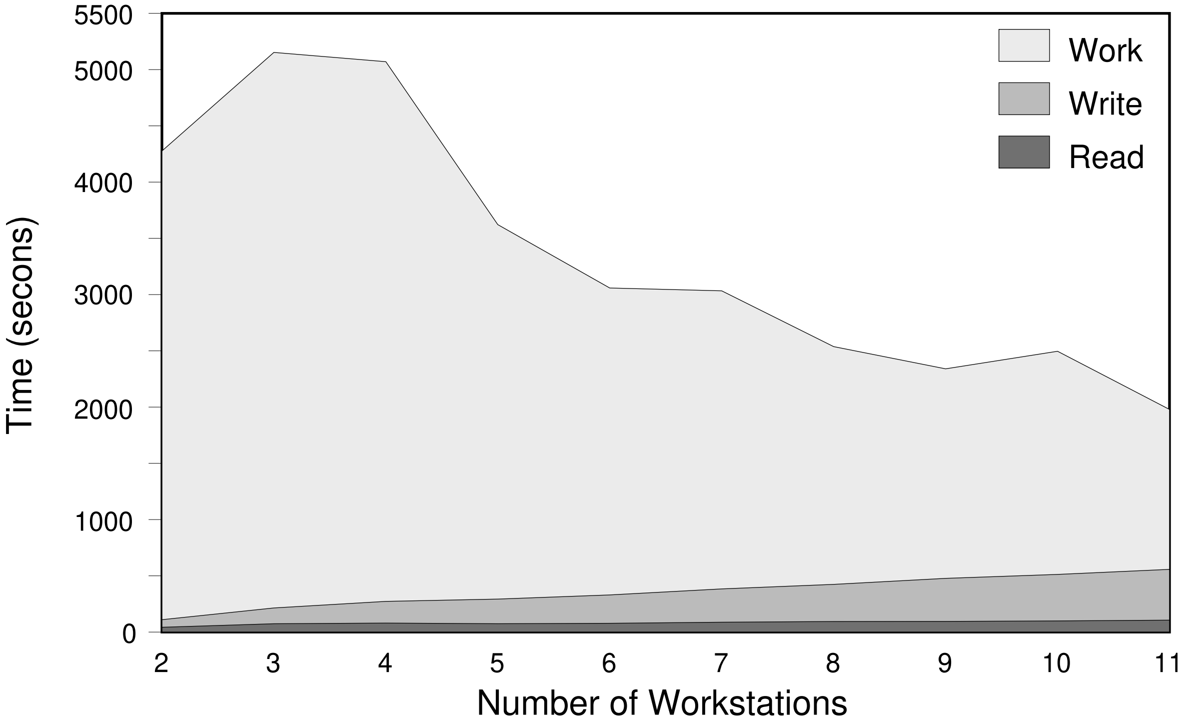

Figure 17: Summary Timing Results (Pruned).

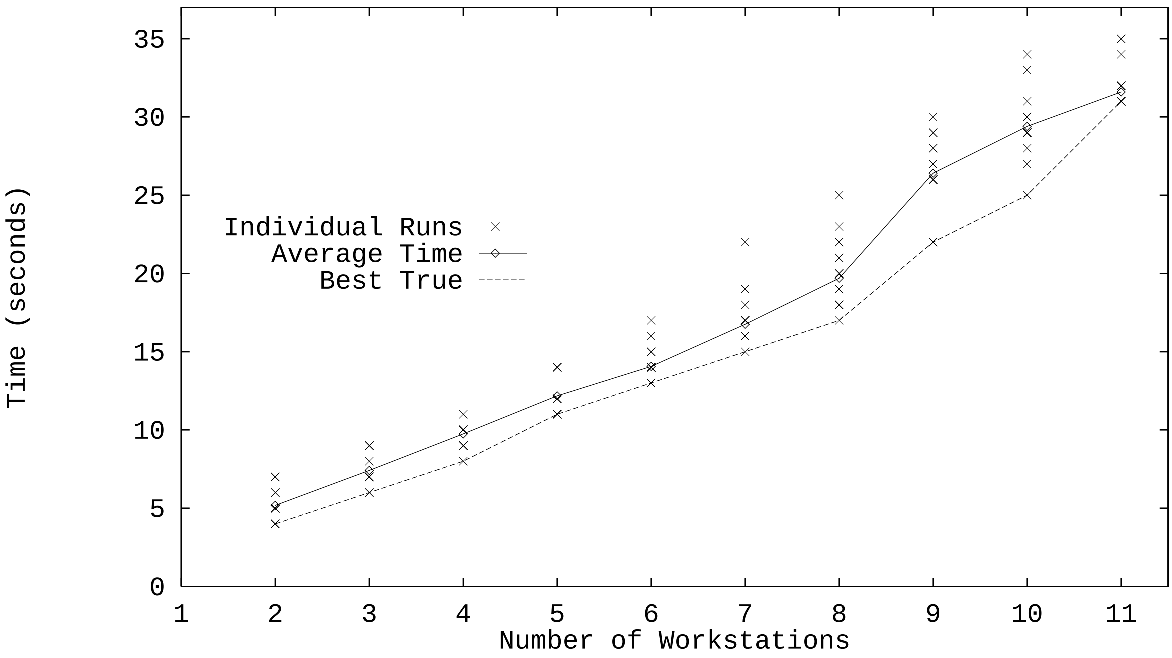

Figure 18: Initialization and Connect Time for the `host` program.

References

[Anderson, 1991] Ray Anderson, "Interprocessor Communication in Dataparallel C," Technical Report 91-80-2, Oregon State University, Corvallis, OR, May 1991.

[Crowl, 1992] Lawrence A. Crowl, "Guidelines for Presenting Parallel Performance," Technical Report 92-80-07, Oregon State University, Corvallis, OR, December 1992.

[Daly *et al.*, 1993] Christopher Daly, Ronald P. Neilson, and Donald L. Phillips, "A Statistical-Topographic Model for Estimating Climatological Precipitation Over Mountainous Terrain," *accepted to Journal of Applied Meteorology*, 1993.

[Flynn, 1966] M. J. Flynn, "Very High-speed Computing Systems," *Proceedngs of the IEEE*, 54(12):1901–1909, December 1966.

[Frew, Jr., 1990] James Edward Frew, Jr., *The Image Processing Workbench*, PhD thesis, University of California, Santa Barbara, July 1990.

[GRASS] United States Army Corps of Engineers — Construction Engineering Research Laboratory, *GRASS User's Reference Manual*, 4.1 edition.

[Hansen *et al.*, 1988] J. Hansen, I. Fung, A. Lacis, D. Rind, S. Lebedeff, and R. Ruedy, "Global Climate Changes as Forcast by Goddard Institute for Space Studies Three-dimensional Model," *Journal of Geophysical Research*, 93:9341–9364, 1988.

[Hansen *et al.*, 1983] J. Hansen, G. Russel, D. Rind, P. Stone, A. Lacis, S. Lebedeff, R. Ruedy, and L. Travis, "Efficient Three-Dimensional Global Models for Climate Studies: Models I and II," *Monthly Weather Review*, 3(4), 1983.

[Hatcher and Quinn, 1991] Philip J. Hatcher and Michael J. Quinn, *Data-Parallel Programming on MIMD Computers*, Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1991.

[Hatcher *et al.*, 1991] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones, "Data-Parallel Progamming in MIMD Computers," *IEEE Transactions on Parallel and Distributed Systems*, 3(2):377–383, July 1991.

[Hatcher and Quinn, 1992] Phillip J. Hatcher and Michael J. Quinn, "Supporting Data-level and Processor-level Parallelism in Data-parallel Programming Languages," Technical Report 92-80-03, Oregon State University, Corvallis, OR, 1992.

[Hillis and Steele, 1986] W. D. Hillis and G. L. Steele, "Data Parallel Algorithms," *Communications of the ACM*, 29(12):1790–1183, December 1986.

[Johnson, 1977] S. C. Johnson, "Lint, a C Program Checker," Computing Science Technical Report CSTR-65, AT&T Bell Laboratories, December 1977, updated version TM 78-1273-3.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., 1978.

[Larcher, 73] Walter Larcher, *Physiological Plant Ecology*, Springer-Verlag, Berlin, second edition, 73, English Translation by Marguerite A. Biederman-Thorson.

[Longley *et al.*, 1992] Kelly Longley, Danna Jacobson, and Danny Marks, "Supplement to the Image Processing Workbench (IPW): Modifications, Procedures, and Software Additions, November 1989 to June 1992 (Revison 2.0)," Technical report, U.S. EPA Environmental Research Laboratory, Corvallis, 1992.

[Longley and Marks, 1991] Kelly Longley and Danny Marks, "Supplement to the Image Processing Workbench (IPW), Volume 1.0, Modification, Procedures, and Software Additions November, 1989 to October, 1991," Technical report, U.S. EPA Environmental Research Laboratory, Corvallis, 1991.

[Manabe and Wetherald, 1987] S. Manabe and R. T. Wetherald, "Large-scale Changes in Soil Wetness Induced by an Increase in Carbon Dioxide," *Journal of Atmospheric Sciences*, 44:1211–1235, April 1987.

[Marks, 1992] Danny Marks, "A Continential-scale Simulation of Potential Evapotranspiration for Historical and Projected Doubled-CO_2 Climate Conditions," *Submitted*, 1992.

[Mithcell and Warrilow, 1987] J. F. B. Mithcell and D. A. Warrilow, "Summer Dryness in Northern Mid Latitudes Due to Increased $\mathrm{CO_2}$," *Nature*, 330(19):238–240, 1987.

[Nedeljkovic and Quinn, 1992] Nenad Nedeljkovic and Michael J. Quinn, "Data-parallel Programming on a Network of Heterogeneous Workstations," Technical Report 91-80-02, Oregon State University, Corvallis, OR, April 1992.

[Neilson *et al.*, 1989] R. P. Neilson, G. A. King, R. L. DeVelice, J. Lenihan, D. Marks, J. Dolph, W. Campbell, and G. Glick, "Sensitivity of Ecological Landscapes for Global Climatic Change," Technical Report EPA-600-3-89-073, NTIS-PB-90-120-072-AS, United States Environmental Protection Agency, 1989.

[Neilson] Ronald P. Neilson, "A Model for Predicting Contential-Scale Vegetation Distribution and Water Balance," *Ecological Applications*, *in revision.*

[Neilson *et al.*, 1992] Ronald P. Neilson, George A. King, and Greg Koerper, "Toward a Rule-based Biome Model," *Landscape Ecology*, 7:27–43, 1992.

[Quinn, 1987] Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Book Company, New York, 1987.

[Quinn and Hatcher, 1990] Michael J. Quinn and Philip J. Hatcher, "Data-Parallel Programming on Multicomputers," *IEEE Software*, pages 69–76, September 1990.

[Rose and Steele, 1987] J. R. Rose and G. L. Steele, "C*: An extended C langauge for data parallel programming," Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[Schlesinger and Zhao, 1989] M. E. Schlesinger and Z. C. Zhao, "Seasonal Climate Change Induced by Double $\mathrm{CO_2}$ as Simulated by the OSU Atmospheric GCM/Mixed-layer Ocean Model," *Journal of Climate*, 2:429–495, 1989.

[Stallman, 1992] Richard M. Stallman, *Using the GNU C Compiler*, Free Software Foundation, May 1992.

[SunOS] Sun Microsystems, Inc., *SunOS Reference Manual.*

[Walter, 1964] Heinrich Walter, *Ecology of Tropical and Suntropical Vegetation*, Oliver & Boyd, Edinburgh, second edition, 1964, English Translation by D. Mueller-Dombois.

[Wetherald and Manabe, 1988] R. T. Wetherald and S. Manabe, "Cloud Feedback Processes in a General Circulation Model," *Journal of Atmospheric Sciences*, 45:1397–1415, April 1988.

Appendix

A Options used with `lint` on the MAPSS program

MAPSS had only a single `line` comment about unused identifier introduced from a system include file.

- `-a`: Warn of assignments of `long` values to variables which are not `long`.
- `-b`: Warn of `break` statements which cannot be reached.
- `-c`: Warn about casts of questionable portability
- `-h`: Apply a number of heuristics to intuit bugs, improve style, and reduce waste,
- `-x`: Report variables referred to by `extern` declarations, but never used.

B Options used with `gcc` on the MAPSS program

- `-Wimplicit`: Warn whenever a function or parameter is implicitly declared.
- `-Wreturn-type`: Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.
- `-Wunused`: Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined, and whenever a statement computes a result that is explicitly not used.
- `-Wswitch`: Warn whenever a `switch` statement has an index of an enumerated type and lacks a `case` for one or more of the named codes of that enumeration. `Case` labels outside the enumeration range also provoke warnings when this option is used.
- `-Wcomment`: Warn whenever a comment-start sequence `/*` appears in a comment.
- `-Wtrigraphs`: Warn if any trigraphs are encountered (assuming they are enabled).
- `-Wformat`: Check calls to `printf` and `scanf`, *etc.*, to make sure that the arguments supplied have types appropriate to the format string specified.
- `-Wchar-subscripts`: Warn if an array subscript has type `char`.
- `-Wuninitialized`: An automatic variable is used without first being initialized.
- `-Wparentheses`: Warn if parentheses are omitted in certain contexts.
- `-Wshadow`: Warn whenever a local variable shadows another local variable.
- `-Wpointer-arith`: Warn about anything that depends on the "size of" a function type or of `void`.
- `-Wcast-qual`: Warn whenever a pointer is cast so as to remove a type qualifier from the target type.
- `-Wcast-align`: Warn whenever a pointer is cast such that the required alignment of the target is increased.
- `-Waggregate-return`: Warn if any functions that return structures or unions are defined or called.

C FTP sites for IPW and GRASS

GRASS may be obtained from moon.cecer.army.mil (129.229.20.254).

IPW may be obtained from cerberus.cor.epa.gov (134.67.4.48).

D Sequential Program Run Times

D.1 Unpruned (seconds reported by `/bin/time`)

3340.3	3337.7	3337.8	3338.8	3396.4	3341.6	3379.7	3352.0	3340.7
3338.1	3341.9	3338.3	3340.3	3339.4	3338.5	3340.0	3340.7	3342.0
3336.5	3338.7	3339.2	3340.8	3338.9				

D.2 Pruned (seconds reported by `/bin/time`)

3320.3	3319.1	3319.0	3321.3	3426.1	3324.6	3361.0	3326.1	3320.0
3317.8	3322.5	3317.8	3318.1	3318.8	3322.6	3323.9	3319.2	3320.2
3321.5	3320.0	3319.2	3318.7	3320.4				

E Time Results for the Dataparallel C MAPSS

E.1 Unpruned Run Times (seconds reported by `/bin/time`)

2	3	4	5	6	7	8	9	10	11
4468.0	3591.5	3647.5	2917.7	1929.7	2475.6	2399.9	2077.5	2218.7	2274.1
4986.1	3522.5	2778.3	2175.4	1860.9	1781.4	1946.7	1708.4	1708.7	1838.5
4352.6	3913.6	2739.2	2312.3	1957.2	2012.8	2349.9	1816.9	2016.9	2193.5
4275.6	3378.0	2569.6	2166.2	1825.0	1916.9	1948.5	1748.7	1650.2	1678.3
4416.9	3922.1	2830.2	2163.8	2067.7	2910.3	3362.0	2470.1	2529.4	2580.9
4284.2	3426.7	2989.2	2481.9	2089.3	1792.7	2184.2	1963.1	1868.1	1813.5
4294.8	3424.9	3018.2	4587.4	3445.3	3043.6	3440.4	2725.9	2032.6	1903.2
7192.7	3794.7	2924.8	2569.9	1900.2	1848.6	2180.9	1905.3	1897.5	1974.2
4284.2	3436.3	2900.2	2380.7	2086.9	1788.0	2198.2	1960.3	1862.5	1801.7
7109.7	3844.1	2936.1	2527.3	1862.1	1810.8	2142.3	1913.8	1859.5	2013.2
4277.7	3423.5	2899.2	2377.1	2082.9	1777.5	2214.2	1918.1	1856.0	1806.0
4288.1	3483.8	3033.5	2469.1	2111.3	1803.5	2216.4	1971.2	1880.5	1829.2
4284.0	3441.1	3023.6	2469.8	2096.3	1778.5	2166.0	1948.7	1849.8	1790.0
4279.4	3423.8	5829.7	4557.3	3551.6	2576.5	2574.2	2051.9	1926.3	1870.7
4266.5	3358.8	2504.4	2155.5	1817.1	2039.6	1940.8	2377.8		
4198.2	3294.6	2493.8	2109.0	1745.8	1857.3	1883.4	1603.4		
5649.8	3338.3	2518.9	2092.6	1981.7	1693.9	1944.3	1639.8		
4198.4	3295.9	2623.2	2596.5	1768.7	1760.7	1882.1	1634.1		
4222.8	11481.7	8791.3	5103.4	3295.4	2136.4	2418.2	2031.2	2533.5	2502.0
4231.2	3370.4	3086.0	2474.5	2047.1	1752.6	2088.8	1878.3	1776.1	1710.9
4223.0	11447.1	8708.1	4992.1	3242.2	2066.2	2349.3	1986.0	1880.0	1839.4
4219.2	3371.0	3104.3	2480.9	2041.0	1711.1	2072.2	1830.3	1748.6	1691.3
4210.1	3591.8	3095.5	2486.6	2039.4	1714.8	2088.4	1830.1	1755.5	1696.8
4343.0	3395.1	3092.0	2621.1	2062.7	1771.3	2149.2	1901.0	1792.8	1731.0
4323.5	3373.8	6293.7	4821.3	3559.1	2634.5	2610.9	1907.4	1832.4	

E.2 Pruned Run Times (seconds reported by `/bin/time`)

2	3	4	5	6	7	8	9	10	11
5653.0	3466.9	3169.3	2568.4	2097.1	2187.7	1846.3	1323.7	1429.9	1195.5
5664.4	3557.8	4155.9	3334.9	2875.2	2660.1	2323.9	1977.0	2171.2	1765.2
5021.9	3108.8	5074.9	4101.5	3426.7	3257.9	2451.9	2329.3	2534.6	1857.8
5441.7	3247.5	5401.0	4454.8	3667.8	3477.1	2999.3	2514.3	2719.0	2180.8
5113.1	3175.2	4272.0	3583.8	2936.5	2788.1	2385.7	2017.1	2205.0	1779.0
5096.0	3101.5	2524.9	1894.6	1546.4	1660.6	1330.3	1153.6	1430.4	1099.6
5082.7	3118.0	2546.7	1884.2	1562.6	1657.5	1337.9	1159.5	1425.5	1098.7
5047.2	3206.7	2780.7	2143.7	2112.7	1508.6	1354.0	3656.2	4109.0	3345.7
5398.8	3262.4	4331.4	3422.5	2946.8	2737.1	2373.2	1999.8	2168.7	1768.2
5055.9	3261.9	4317.6	3430.9	2825.6	2749.6	2358.5	1978.5	2152.0	1761.9
3375.1	11098.6	7222.8	4983.4	4288.5	4073.7	3378.6	2822.6	2897.3	2347.9
3378.2	7723.6	7142.5	5008.8	4280.4	4046.1	3382.2	2838.8	2893.5	2320.0
3377.8	7716.4	7177.1	5021.3	4247.9	4069.4	3392.6	2814.0	2896.2	2328.5
3375.8	7723.8	7202.1	4992.5	4276.4	4038.8	3367.4	2855.7	2947.0	2152.9
3376.7	7829.3	6980.8	4940.9	4215.1	3992.9	3309.1	2787.6	2833.1	2443.8
3556.0	4606.2	3976.6			2557.1	2188.6	1761.5	1825.6	1533.1
3550.4	2307.0	2072.9	1726.9	1428.7	1432.8	1427.9	1325.2	1426.9	1055.5
3554.8	7428.6	6758.9	4718.0	4024.4	3909.7	3197.7	3730.1	4200.2	3425.7
3561.2	7383.8	6763.5	4755.9	4031.7	3863.6	3186.8	2663.7	2719.1	2208.1
3559.1	7328.2	6730.7	4736.8	4016.2	3887.4	3158.3	2633.6	2757.1	2210.3
3486.3	2299.0	4181.4	3410.8	2732.2	2764.9	2340.6	3010.7	3178.6	2181.2
3516.3	7622.0	7044.6	4918.1	4144.6	3945.5	3264.1	2778.6	2811.1	2312.6

E.3 Unpruned Read Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
82	77	94	116	126	147	154	163	181	193
85	79	99	111	128	138	148	163	179	193
87	86	97	113	128	143	153	165	179	200
82	83	96	116	126	145	145	166	180	195
80	85	98	114	131	144	154	165	180	190
82	80	93	111	128	137	151	162	174	190
85	85	97	114	132	146	153	164	182	196
81	82	97	108	127	143	152	164	170	183
85	85	98	113	124	141	146	160	171	183
37	96	84	87	83	92	96	88	90	91
59	134	138	142	135	139	130	136	139	128
60	60	68	79	86	98	105	113	121	131
59	135	137	136	141	131	136	137	128	138
56	61	66	77	88	98	104	111	119	128
51	97	67	77	89	97	101	112	120	129
55	66	66	77	86	98	105	110	118	130
58	57	94	96	100	104	104	110	123	
56	61	68	81	93	104	108	117	126	135

E.4 Pruned Read Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
61	70	74	83	88	102	97	114	128	130
60	62	69	85	93	102	112	118	125	131
51	56	71	75	80	90	99	103	112	120
63	67	72	78	107	87	98	102	110	116
51	58	74	71	77	86	97	98	106	112
51	58	63	68	72	85	94	92	105	109
50	60	63	70	74	87	95	94	93	109
46	59	62	74	79	88	99	117	122	125
61	55	66	73	79	90	97	102	108	117
50	55	61	70	72	96	92	95	104	110
30	107	210	87	107	92	97	89	87	91
30	99	87	86	85	92	97	89	86	90
31	99	120	89	86	91	99	89	88	92
30	100	88	88	87	92	97	90	89	136
30	98	85	86	85	91	96	88	89	91
36	63	56			71	87	66	74	95
34	42	45	50	55	62	68	70	78	82
34	96	84	85	81	89	94	117	119	120
35	95	81	84	82	90	95	86	89	88
35	95	83	83	84	90	92	84	86	102
35	41	63	63	60	64	68	96	96	77
37	96	84	87	83	92	96	88	90	91

E.5 Unpruned Write Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
131	196	238	299	353	411	494	525	583	629
136	186	252	341	416	454	506	566	627	667
148	224	253	339	377	447	495	511	614	615
153	221	273	295	345	406	468	519	576	622
137	189	238	298	358	413	509	524	580	624
124	179	243	303	378	406	530	523	582	627
137	199	268	308	363	420	482	536	595	643
134	196	256	297	348	408	483	515	574	619
124	183	287	356	406	462	539	589	647	698
58	167	212	237	265	309	341	429	429	484
95	273	353	409	465	513	600	658	641	691
97	156	213	265	323	381	454	492	549	594
94	252	322	371	443	482	556	617	655	713
93	153	215	262	319	379	439	482	532	576
94	156	211	262	315	378	433	483	538	583
94	153	207	260	318	389	453	511	596	612
93	153	270	373	384	437	500	557	612	
104	161	235	289	336	396	461	505	564	607

E.6 Pruned Write Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
96	156	185	253	261	279	352	341	366	412
87	122	199	231	266	288	333	389	411	449
77	111	200	244	264	303	342	366	401	416
84	121	213	260	282	333	394	423	464	480
75	124	195	227	321	295	323	370	385	422
74	104	140	172	204	244	276	305	337	373
73	105	141	172	205	244	275	306	341	373
77	119	161	192	218	251	280	429	456	491
74	122	191	212	244	279	314	373	379	412
73	106	177	213	284	280	312	347	378	412
55	273	234	272	302	341	373	407	442	483
56	161	236	271	304	335	380	408	440	478
56	166	237	263	302	342	369	405	442	477
57	163	236	270	303	340	372	421	467	480
56	163	231	265	303	341	370	413	445	621
57	120	155			297	285	316	348	381
60	97	134	165	203	231	266	306	330	362
60	166	203	234	266	300	338	504	542	574
60	163	203	233	265	301	337	375	401	444
61	166	200	235	263	304	342	371	469	451
55	90	157	195	222	276	287	413	418	454
58	167	212	237	265	309	341	429	429	484

E.7 Unpruned Work Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
4060	3145	2645	2050	1593	1205	1510	1240	1071	955
4062	3150	2654	4118	2873	2429	2763	1968	1191	1008
6947	3468	2562	2101	1373	1240	1508	1201	1072	1101
6867	3530	2556	2102	1373	1243	1506	1201	1072	1153
4062	3152	2553	1955	1583	1206	1510	1239	1072	955
4063	3155	2551	1948	1560	1206	1507	1205	1072	955
4058	3190	2655	2032	1601	1206	1556	1239	1072	955
4061	3154	2659	2049	1606	1204	1504	1236	1073	954
4063	3146	5432	4073	3002	1950	1869	1263	1071	955
3415	7350	6736	4576	3777	3523	2806	2232	2257	1691
4061	11065	8281	4535	2676	1464	1666	1204	1714	1641
4066	3145	2792	2117	1620	1248	1508	1245	1072	953
4061	11050	8233	4469	2639	1434	1636	1203	1065	954
4065	3147	2812	2129	1618	1209	1504	1204	1064	954
4059	3328	2806	2136	1619	1208	1534	1203	1065	953
4063	3413	2709	2076	1621	1253	1521	1252	1065	956
4165	3155	5918	4336	3052	2073	1985	1206	1065	
4175	3162	2777	2234	1615	1249	1558	1244	1071	955

E.8 Pruned Work Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
5487	3227	2894	2215	1723	1782	1371	839	897	616
5506	3361	3874	3002	2496	2246	1857	1440	1577	1145
4883	2932	4789	3763	3064	2841	1986	1811	1986	1286
5286	3045	5103	4101	3259	3033	2483	1962	2112	1546
4978	2983	3989	3271	2517	2383	1945	1520	1665	1208
4964	2931	2310	1642	1257	1314	935	729	959	580
4954	2945	2330	1626	1269	1305	944	733	962	580
4917	3019	2545	1863	1798	1145	949	3080	3495	2695
5255	3073	4063	3122	2605	2344	1941	1501	1634	1200
4924	3091	4067	3134	2450	2347	1933	1507	1638	1201
3283	10709	6759	4607	3853	3619	2887	2299	2333	1741
3287	7456	6809	4634	3875	3598	2886	2310	2336	1720
3286	7444	6806	4653	3842	3615	2904	2294	2337	1726
3283	7452	6866	4617	3870	3586	2879	2312	2351	1497
3284	7559	6654	4574	3808	3540	2824	2258	2258	1689
3457	4415	3753			2168	1769	1345	1371	1016
3449	2159	1884	1498	1153	1119	1067	919	968	571
3453	7157	6460	4379	3659	3478	2741	3086	3506	2697
3460	7117	6468	4418	3668	3446	2732	2174	2199	1642
3457	7060	6437	4401	3652	3460	2700	2152	2168	1615
3390	2159	3950	3139	2431	2402	1961	2473	2632	1613
3415	7350	6736	4576	3777	3523	2806	2232	2257	1691

E.9 Connect Times (seconds reported by `time()`)

2	3	4	5	6	7	8	9	10	11
6	9	10	14	16	18	22	29	33	35
5	7	9	11	14	16	20	28	25	31
5	6	10	12	15	17	23	28	29	31
7	9	10	14	14	22	25	30	34	35
6	9	10	14	15	17	21	29	30	32
7	8	10	14	15	19	21	22	31	32
7	9	11	14	17	19	22	29	27	34
4	7	10	12	14	17	20	22	30	32
5	7	10	12	14	17	18	26	29	31
5	7	10	12	14	16	19	26	29	31
4	7	10	11	13	16	18	26	29	32
5	7	9	11	14	16	18	26	29	31
4	7	10	11	14	16	20	22	30	31
5	7	9	12	13	17	19	26	29	31
4	7	9	11	13	16	17	26	29	32
4	6	8	11	13	15	19	27	29	32
5	7	9	11	14	16	18	27	28	31

F Dataparallel C Code in MAPSS

F.1 Input code for the Dataparallel C Compiler

Full Domain Declaration

```
# define YEAR       12

domain Datum {
    float           sum_ppt,
                    sum_pet,
                    sum_tmp,
                    sum_snow;
    float           lai[YEAR][2],
                    rai[YEAR][2],
                    grass_lai[YEAR],
                    conductance[YEAR][2];
    float           maxlai,
                    begin_maxlai,
                    offset,
                    max_ppt,
                    growing_deg_days;
    int             mask,
                    shrub;
    Boolean         did_exceed_at,
                    exceeded_at_thresh;
    DataInputs      data_point;
    DataOutputs     data_calc;
    Rulecat         cats;
    Month           month[YEAR];
    State           site[YEAR + YEAR],
                    *this_state,
                    *prev_state;
};


# ifndef LINEAR
#  ifdef TINY
#   define NUM_ROWS 3
#   define NUM_COLS 2
#   define DEFAULT_INPUT        \
        "/files/data/terra/data1/jesse/src/dpc_data/tiny.asc"
#  endif /* TINY */

#  ifdef SMALL
#   define NUM_ROWS 11
#   define NUM_COLS 51
#   define DEFAULT_INPUT        \
        "/files/data/terra/data1/jesse/src/dpc_data/small.asc"
#  endif /* SMALL */

#  ifdef MEDIUM
#   define NUM_ROWS 51
#   define NUM_COLS 201
#   define DEFAULT_INPUT        \
        "/files/data/terra/data1/jesse/src/dpc_data/medium.asc"
#  endif /* MEDIUM */

#  ifdef BIG
#   define NUM_ROWS 295
#   define NUM_COLS 470
#   define DEFAULT_INPUT        \
        "/files/data/terra/data1/jesse/src/dpc_data/big.asc"
```

```
#  endif /* BIG */

domain Datum        DP[NUM_ROWS][NUM_COLS];

#  define Self      (this - &DP[0][0])

# else /* LINEAR */

#  ifdef TINY
#   define NUM_VPS  6
#   define DEFAULT_INPUT       \
        "/files/data/terra/data1/jesse/src/dpc_data/tiny.prune"
#  endif /* TINY */

#  ifdef SMALL
#   define NUM_VPS  345
#   define DEFAULT_INPUT       \
        "/files/data/terra/data1/jesse/src/dpc_data/small.prune"
#  endif /* SMALL */

#  ifdef MEDIUM
#   define NUM_VPS  5623
#   define DEFAULT_INPUT       \
        "/files/data/terra/data1/jesse/src/dpc_data/medium.prune"
#  endif /* MEDIUM */

#  ifdef BIG
#   define DEFAULT_INPUT       \
        "/files/data/terra/data1/jesse/src/dpc_data/big.asc"
#  endif /* BIG */

domain Datum        DP[NUM_VPS];

#  define Self      (this - &DP[0])

# endif /* LINEAR */
```

Principle MAPSS Routine

```
void
station_cycle(all_files)
    char            *all_files[];
{
    extern float    tundra_boundry;
    extern float    taiga_tundra_boundry;
    extern time_t   begin_read,
                    begin_process,
                    end_process,
                    end_write;

    FILE            *input;

    begin_read = time(NULL);
    input = dpc_fopen(DEFAULT_INPUT, "r");
    read_input(input);
    begin_process = time(NULL);

    [domain Cell].{         /* station cycle */
        float       max_ppt,
                    growing_deg_days;
        int         mo,
                    id;
```

```
        id = Self;
#ifndef LINEAR
        data_point.row = id / NUM_COLS;
        data_point.col = id % NUM_COLS;
#endif /* LINEAR */
        data_calc.neighbor_diff = FALSE;
        if (mask) {
            /*****
                This loop will terminate when all the stations have
                been read and processed.  The stations are read in
                the routine read_input().  It will exit upon EOF.
            ******/
            for (mo = JAN; mo <= DEC; mo++) {
                sum_ppt += data_point.ppt[mo];
                sum_pet += data_point.pet[mo];
                sum_tmp += data_point.tmp[mo];
            }
            if ((sum_ppt == 0.0) || (sum_pet == 0.0) || (sum_tmp == 0.0)) {
                data_calc.tree_lai =
                data_calc.shrub_lai =
                data_calc.avg_grass_lai =
                data_calc.total_grass_lai = 0.0;

                data_calc.wood_at =
                data_calc.grass_at =
                data_calc.max_snow =
                data_calc.hi_runoff =
                data_calc.lo_runoff = 0.0;

                data_calc.mo_hi_runoff = UNKNOWN;

                data_calc.rtop =
                data_calc.tot_runnoff = 0;

                cats.deciduous =
                cats.broadleaf =
                cats.grassfire = FALSE;

                cats.thaw = SpringThaw(data_point.tmp);
                cats.StationClass = OutOfBounds;
            } else {
                /* set lai iteration cycle site pointers, search mode */
                this_state = site;
                prev_state = site + YEAR;
                did_exceed_at = FALSE;

                exceeded_at_thresh = FALSE;

                /********
                    This is a kludge.  These data points were causing the
                    model to spin and never reach a solution.  So for now
                    I simply output them as hot desert and move along.
                *********/
                for (mo = JAN; mo <= DEC; mo ++) {
                    if (data_point.ppt[mo] > max_ppt) {
                        max_ppt = data_point.ppt[mo];
                    }
                }
                if (max_ppt <= PPT_CUTOFF) {
                    data_calc.tree_lai =
                    data_calc.shrub_lai =
                    data_calc.avg_grass_lai =
                    data_calc.total_grass_lai = 0.0;
```

```
            data_calc.wood_at =
            data_calc.grass_at =
            data_calc.max_snow =
            data_calc.hi_runoff =
            data_calc.lo_runoff = 0.0;

            data_calc.mo_hi_runoff = UNKNOWN;

            data_calc.rtop =
            data_calc.tot_runnoff = 0;

            cats.deciduous =
            cats.broadleaf =
            cats.grassfire = FALSE;

            cats.thaw = SpringThaw(data_point.tmp);
            cats.StationClass = HotDesert;
        }
        else {
            if (SouthernHemi) {
                SouthernHemiSwap(data_point.ppt, data_point.pet,
                    data_point.tmp);
            }

            mapss_new_station(data_point.ppt, data_point.pet,
                data_point.tmp, month);

            /* apriori rules catagorization */
            apriori_catagorization(&data_point, month, &cats);
                growing_deg_days = GrowingDegreeDays(data_point.tmp,
                GROWING_DEG_BASE);
            if (growing_deg_days <= taiga_tundra_boundry) {
                int             mo;
                float           ann_ppt = 0.0,
                                mo_runoff;

                data_calc.tree_lai =
                data_calc.shrub_lai =
                data_calc.avg_grass_lai =
                data_calc.total_grass_lai = 0.0;

                data_calc.wood_at =
                data_calc.grass_at =
                data_calc.max_snow =
                data_calc.hi_runoff = 0.0;
                data_calc.lo_runoff = MAXFLOAT;

                data_calc.mo_hi_runoff = UNKNOWN;

                cats.grassfire = FALSE;
                cats.StationClass =
                    (growing_deg_days <= tundra_boundry) ?
                    Tundra : TaigaTundra;
                for (mo = JAN; mo <= DEC; mo++) {
                    ann_ppt += data_point.ppt[mo];
                    if (month[mo].snow > data_calc.max_snow) {
                        data_calc.max_snow = month[mo].snow;
                    }
                    data_calc.ann_runoff +=
                        (mo_runoff = month[mo].melt_rate *
                        month[mo].snow);
                    if (mo_runoff > data_calc.hi_runoff) {
                        data_calc.hi_runoff = mo_runoff;
                        data_calc.mo_hi_runoff = mo;
```

```
                            }
                            if (mo_runoff < data_calc.lo_runoff) {
                                data_calc.lo_runoff = mo_runoff;
                            }
                        }
                        data_calc.rtop =
                            (int) (100.0 * data_calc.ann_runoff / ann_ppt);
                        data_calc.tot_runnoff =
                            (int) (data_calc.ann_runoff + 0.5);
                    }
                    else {
                        begin_maxlai = maxlai =
                            input_leaf_root_areas(lai, rai,
                                data_point.tmp, &offset, grass_lai,
                                cats.deciduous && cats.broadleaf);

                        /*
                            calculate storm event frequency, melt
                            rate, etc
                        */
                        snow_and_melt(month, data_point.tmp,
                            data_point.ppt);
                        rainfall_events(month, data_point.ppt,
                            data_point.pet);
                            /* Iterate through lai cycle */
                        lai_iterate_cycle(&begin_maxlai, &maxlai,
                            this_state, prev_state, lai, month,
                            grass_lai, &offset,
                            &cats, &data_point, &data_calc);
                        if (did_exceed_at) {
                            total_final_exceed = total_final_exceed + 1;
                        }
                    }
                }
            }
        }
        else {
            /*****
                This is data excluded by the mask.  It is water,
                Canada, or Mexico.
            ******/
            data_calc.tree_lai =
            data_calc.shrub_lai =
            data_calc.avg_grass_lai =
            data_calc.total_grass_lai = 0.0;

            data_calc.wood_at =
            data_calc.grass_at =
            data_calc.max_snow =
            data_calc.hi_runoff =
            data_calc.lo_runoff = 0.0;

            data_calc.mo_hi_runoff = UNKNOWN;

            data_calc.rtop =
            data_calc.tot_runnoff = 0;

            cats.deciduous =
            cats.broadleaf =
            cats.grassfire = FALSE;

            cats.thaw = UNKNOWN;
            cats.StationClass = Unknown;
        }
```

```
    } /* [domain Cell].{ */

    end_process = time(NULL);
    DoOutput();
    end_write = time(NULL);
}
```

F.2 Output from the Dataparallel C Compiler

Principle MAPSS Routine

```
/*
**  Dataparallel C Translator, version 0.95
**  Last complete build: 01:10:05 Oct 14 1992
**  Philip Hatcher, Robert Jones, and Anthony Lapadula
**  Copyright (c) 1990, 1991 University of New Hampshire
**  Questions/comments/etc. should be sent to Anthony Lapadula
**      at al@cs.unh.edu or ..!uunet!unhd!al
**  File translated Mon Dec 14 19:03:19 1992
**  Command used for initial pass through preprocessor:
**      "/lib/cpp  -I/usr/include dpc_mapssCPP.dpc"
**  Input file:  dpc_mapssCPP.dpc
**  Output file: dpc_mapssCPP.c
**  Expecting 11 physical processors
*/

void
station_cycle (all_files)
  char *(*all_files);
{
  register int DPC_vpi = 0; /* VP-emulation loop counter */

  extern float tundra_boundry;
  extern float taiga_tundra_boundry;
  extern long begin_read;
  extern long begin_process;
  extern long end_process;
  extern long end_write;
  struct _iobuf (*input);

  ;             /* Just skipped this tree:  <EMPTY EXPRESSION> */
  begin_read = time(NULL);
  input = dpc_fopen ("/files/data/terra/data1/jesse/src/dpc_data/medium.prune", "r");
  read_input (input);
  begin_process = time(NULL);

  _polyprint = 1;

  /* [domain Cell]. */
  {
    short DPC_temp_38[512] /* 'if' cond. */ ;
    short DPC_temp_37[512] /* 'else' cond. */ ;
    short DPC_temp_36[512] /* 'if' cond. */ ;
    short DPC_temp_35[512] /* 'else' cond. */ ;
    short DPC_temp_34[512] /* 'if' cond. */ ;
    short DPC_temp_33[512] /* 'else' cond. */ ;
    short DPC_temp_32[512] /* 'if' cond. */ ;
    short DPC_temp_31[512] /* 'else' cond. */ ;
    short DPC_temp_30[512] /* 'if' cond. */ ;

    {
      float max_ppt[512];
```

```
    float growing_deg_days[512];
    int mo[512];
    int id[512];

    ;            /* Just skipped this tree:  <EMPTY EXPRESSION> */

    for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
  {
    {
      id[DPC_vpi] = (DPC_get_index_Cell (DPC_nodenum, DPC_vpi) - 0);
      DP[DPC_vpi].data_calc.neighbor_diff = 0;
      DPC_temp_30[DPC_vpi] = 1;

      if (DPC_temp_30[DPC_vpi])
        {
      DPC_temp_30[DPC_vpi] = ((DP[DPC_vpi].mask) != 0);
        }
      else;      /* Just skipped this tree:  <EMPTY STATEMENT> */
    }
  }
    {
  {

    for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
      {
        {

      if (DPC_temp_30[DPC_vpi])
        {
          {

            for (mo[DPC_vpi] = 0; mo[DPC_vpi] <= 11; mo[DPC_vpi]++)
          {
            {
              DP[DPC_vpi].sum_ppt += DP[DPC_vpi].data_point.ppt[mo[DPC_vpi]];
              DP[DPC_vpi].sum_pet += DP[DPC_vpi].data_point.pet[mo[DPC_vpi]];
              DP[DPC_vpi].sum_tmp += DP[DPC_vpi].data_point.tmp[mo[DPC_vpi]];
            }
          }
            DPC_temp_32[DPC_vpi] = (DPC_temp_30[DPC_vpi]);

            if (DPC_temp_32[DPC_vpi])
          {
            DPC_temp_32[DPC_vpi] = ((((double) (DP[DPC_vpi].sum_ppt)) == 0.0) ||
               (((double) (DP[DPC_vpi].sum_pet)) == 0.0) ||
                (((double) (DP[DPC_vpi].sum_tmp)) == 0.0));
          }
            else; /* Just skipped this tree:  <EMPTY STATEMENT> */
            {

          if (DPC_temp_32[DPC_vpi])
            {
              {
                {

              if (DPC_temp_32[DPC_vpi])
                {
                  {
                    DP[DPC_vpi].data_calc.tree_lai =
                      DP[DPC_vpi].data_calc.shrub_lai =
                      DP[DPC_vpi].data_calc.avg_grass_lai =
                      DP[DPC_vpi].data_calc.total_grass_lai = 0.0;
                    DP[DPC_vpi].data_calc.wood_at =
                      DP[DPC_vpi].data_calc.grass_at =
```

```
                  DP[DPC_vpi].data_calc.max_snow =
                  DP[DPC_vpi].data_calc.hi_runoff =
                  DP[DPC_vpi].data_calc.lo_runoff = 0.0;
                DP[DPC_vpi].data_calc.mo_hi_runoff = -1;
                DP[DPC_vpi].data_calc.rtop =
                  DP[DPC_vpi].data_calc.tot_runnoff = 0;
                DP[DPC_vpi].cats.deciduous =
                  DP[DPC_vpi].cats.broadleaf =
                  DP[DPC_vpi].cats.grassfire = FALSE;
                DP[DPC_vpi].cats.thaw = SpringThaw (DP[DPC_vpi].data_point.tmp);
                DP[DPC_vpi].cats.StationClass = -2;

                ; /* Just skipped this tree:  <EMPTY EXPRESSION> */
              }
            }
          else;   /* Empty else */
            }
          }
        }
      else;   /* Empty else */
        }
      }
    }
  else;       /* Empty else */
    }
  }
{

  for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
    {
  {

    if (DPC_temp_30[DPC_vpi])
      {
        {
      DPC_temp_33[DPC_vpi] = !((int) (DPC_temp_32[DPC_vpi])) && DPC_temp_30[DPC_vpi];
        }
      }
    else;     /* Empty else */
  }
    }
  {
    {

  for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
    {
      {

        if (DPC_temp_30[DPC_vpi])
      {
        {

          if (DPC_temp_33[DPC_vpi])
            {
          {
            DP[DPC_vpi].this_state = DP[DPC_vpi].site;
            DP[DPC_vpi].prev_state = DP[DPC_vpi].site + 12;
            DP[DPC_vpi].did_exceed_at = 0;
            DP[DPC_vpi].exceeded_at_thresh = 0;

            for (mo[DPC_vpi] = 0; mo[DPC_vpi] <= 11; mo[DPC_vpi]++)
              {
                {

```

```
            if (DP[DPC_vpi].data_point.ppt[mo[DPC_vpi]] > max_ppt[DPC_vpi])
              {
                {
                  max_ppt[DPC_vpi] = DP[DPC_vpi].data_point.ppt[mo[DPC_vpi]];
                }
              }
            else;   /* Empty else */
              }
            }
          DPC_temp_34[DPC_vpi] = (DPC_temp_33[DPC_vpi]);

          if (DPC_temp_34[DPC_vpi])
            {
              DPC_temp_34[DPC_vpi] = (((double) (max_ppt[DPC_vpi])) <= 6.0);
            }
          else; /* Just skipped this tree:  <EMPTY STATEMENT> */
          {

            if (DPC_temp_34[DPC_vpi])
              {
            {
              {

                if (DPC_temp_34[DPC_vpi])
                  {
                {
                  DP[DPC_vpi].data_calc.tree_lai =
                    DP[DPC_vpi].data_calc.shrub_lai =
                    DP[DPC_vpi].data_calc.avg_grass_lai =
                    DP[DPC_vpi].data_calc.total_grass_lai = 0.0;
                  DP[DPC_vpi].data_calc.wood_at =
                    DP[DPC_vpi].data_calc.grass_at =
                    DP[DPC_vpi].data_calc.max_snow =
                    DP[DPC_vpi].data_calc.hi_runoff =
                    DP[DPC_vpi].data_calc.lo_runoff = 0.0;
                  DP[DPC_vpi].data_calc.mo_hi_runoff = -1;
                  DP[DPC_vpi].data_calc.rtop =
                    DP[DPC_vpi].data_calc.tot_runnoff = 0;
                  DP[DPC_vpi].cats.deciduous =
                    DP[DPC_vpi].cats.broadleaf =
                    DP[DPC_vpi].cats.grassfire = FALSE;
                  DP[DPC_vpi].cats.thaw = SpringThaw(DP[DPC_vpi].data_point.tmp);
                  DP[DPC_vpi].cats.StationClass = 16;

                  ; /* Just skipped this tree:  <EMPTY EXPRESSION> */
                }
                  }
                else;   /* Empty else */
              }
            }
              }
            else;   /* Empty else */
          }
        }
          }
        else;   /* Empty else */
      }
    }
      else; /* Empty else */
    }
  }
{

  for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
```

```
  {
    {

  if (DPC_temp_30[DPC_vpi])
    {
      {

        if (DPC_temp_33[DPC_vpi])
      {
        {
          DPC_temp_35[DPC_vpi] = !((int) (DPC_temp_34[DPC_vpi])) && DPC_temp_33[DPC_vpi];
        }
      }
        else; /* Empty else */
      }
    }
  else;   /* Empty else */
    }
  }
{
  {

    for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
  {
    {

      if (DPC_temp_30[DPC_vpi])
        {
      {

        if (DPC_temp_33[DPC_vpi])
          {
            {

          if (DPC_temp_35[DPC_vpi])
            {
              {

                if (SouthernHemi)
              {
                {
                  SouthernHemiSwap(DP[DPC_vpi].data_point.ppt,
                    DP[DPC_vpi].data_point.pet, DP[DPC_vpi].data_point.tmp);
                }
              }
                else; /* Empty else */
                mapss_new_station (DP[DPC_vpi].data_point.ppt,
                  DP[DPC_vpi].data_point.pet, DP[DPC_vpi].data_point.tmp,
                  DP[DPC_vpi].month);
                apriori_catagorization (&DP[DPC_vpi].data_point,
                  DP[DPC_vpi].month, &DP[DPC_vpi].cats);
                growing_deg_days[DPC_vpi] = GrowingDegreeDays(
                  DP[DPC_vpi].data_point.tmp, 0.0);
                DPC_temp_36[DPC_vpi] = (DPC_temp_35[DPC_vpi]);

                if (DPC_temp_36[DPC_vpi])
              {
                DPC_temp_36[DPC_vpi] = (growing_deg_days[DPC_vpi] <= taiga_tundra_boundry);
              }
                else; /* Just skipped this tree:  <EMPTY STATEMENT> */
                {

              if (DPC_temp_36[DPC_vpi])
                {
```

```
{
  {
int mo;
float ann_ppt;
float mo_runoff;

if (DPC_temp_36[DPC_vpi])
  {
    {
      {

    if (DPC_temp_36[DPC_vpi])
      {
        {
          ann_ppt = 0.0;
        }
      }
    else;   /* Empty else */
      }
      DP[DPC_vpi].data_calc.tree_lai =
        DP[DPC_vpi].data_calc.shrub_lai =
        DP[DPC_vpi].data_calc.avg_grass_lai =
        DP[DPC_vpi].data_calc.total_grass_lai = 0.0;
      DP[DPC_vpi].data_calc.wood_at =
        DP[DPC_vpi].data_calc.grass_at =
        DP[DPC_vpi].data_calc.max_snow =
        DP[DPC_vpi].data_calc.hi_runoff = 0.0;
      DP[DPC_vpi].data_calc.lo_runoff = ((float) MAXFLOAT);
      DP[DPC_vpi].data_calc.mo_hi_runoff = -1;
      DP[DPC_vpi].cats.grassfire = 0;
      DP[DPC_vpi].cats.StationClass =
        (growing_deg_days[DPC_vpi] <= tundra_boundry) ? 17 : 18;

      for (mo = 0; mo <= 11; mo++)
    {
      {
        ann_ppt += DP[DPC_vpi].data_point.ppt[mo];

        if (DP[DPC_vpi].month[mo].snow > DP[DPC_vpi].data_calc.max_snow)
          {
        {
          DP[DPC_vpi].data_calc.max_snow = DP[DPC_vpi].month[mo].snow;
        }
          }
        else;   /* Empty else */
        DP[DPC_vpi].data_calc.ann_runoff +=
          (mo_runoff = DP[DPC_vpi].month[mo].melt_rate *
          DP[DPC_vpi].month[mo].snow);

        if (mo_runoff > DP[DPC_vpi].data_calc.hi_runoff)
          {
        {
          DP[DPC_vpi].data_calc.hi_runoff = mo_runoff;
          DP[DPC_vpi].data_calc.mo_hi_runoff = mo;
        }
          }
        else;   /* Empty else */

        if (mo_runoff < DP[DPC_vpi].data_calc.lo_runoff)
          {
        {
          DP[DPC_vpi].data_calc.lo_runoff = mo_runoff;
        }
          }
```

```
                        else;   /* Empty else */
                      }
                    }
                      DP[DPC_vpi].data_calc.rtop =
                        (int) (100.0 * ((double) (DP[DPC_vpi].data_calc.ann_runoff)) /
                        ((double) (ann_ppt)));
                      DP[DPC_vpi].data_calc.tot_runnoff =
                        (int) (((double) (DP[DPC_vpi].data_calc.ann_runoff)) + 0.5);

                      ; /* Just skipped this tree:  <EMPTY EXPRESSION> */
                    }
                  }
                else;   /* Empty else */
                  }
                }
              }
            else;   /* Empty else */
              }
            }
          }
        else;   /* Empty else */
          }
        }
      else; /* Empty else */
    }
      }
    else;   /* Empty else */
  }
}
  {

for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
  {
    {

      if (DPC_temp_30[DPC_vpi])
    {
      {

        if (DPC_temp_33[DPC_vpi])
          {
        {

          if (DPC_temp_35[DPC_vpi])
            {
              {
            DPC_temp_37[DPC_vpi] =
              !((int) (DPC_temp_36[DPC_vpi])) && DPC_temp_35[DPC_vpi];
              }
            }
          else; /* Empty else */
        }
          }
        else;   /* Empty else */
      }
    }
      else; /* Empty else */
    }
  }
{
  {

    for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
      {
```

```
                {

                  if (DPC_temp_30[DPC_vpi])
                    {
                      {

                    if (DPC_temp_33[DPC_vpi])
                      {
                        {

                          if (DPC_temp_35[DPC_vpi])
                        {
                          {

                            if (DPC_temp_37[DPC_vpi])
                              {
                            {
                              DP[DPC_vpi].begin_maxlai =
                                DP[DPC_vpi].maxlai =
                                  input_leaf_root_areas (DP[DPC_vpi].lai, DP[DPC_vpi].rai,
                                    DP[DPC_vpi].data_point.tmp, &DP[DPC_vpi].offset,
                                    DP[DPC_vpi].grass_lai,
                                    DP[DPC_vpi].cats.deciduous && DP[DPC_vpi].cats.broadleaf);
                              snow_and_melt (DP[DPC_vpi].month, DP[DPC_vpi].data_point.tmp,
                                DP[DPC_vpi].data_point.ppt);
                              rainfall_events (DP[DPC_vpi].month, DP[DPC_vpi].data_point.ppt,
                                DP[DPC_vpi].data_point.pet);
                              lai_iterate_cycle (&DP[DPC_vpi].begin_maxlai, &DP[DPC_vpi].maxlai,
                                DP[DPC_vpi].this_state, DP[DPC_vpi].prev_state, DP[DPC_vpi].lai,
                                DP[DPC_vpi].month, DP[DPC_vpi].grass_lai, &DP[DPC_vpi].offset,
                                &DP[DPC_vpi].cats, &DP[DPC_vpi].data_point, &DP[DPC_vpi].data_calc);
                              DPC_temp_38[DPC_vpi] = (DPC_temp_37[DPC_vpi]);
                            }
                              }
                            else;   /* Empty else */

                            if (DPC_temp_37[DPC_vpi])
                              {
                            {

                              if (DPC_temp_38[DPC_vpi])
                                {
                                  DPC_temp_38[DPC_vpi] = ((DP[DPC_vpi].did_exceed_at) != 0);
                                }
                              else; /* Empty else */
                            }
                              }
                            else;   /* Empty else */
                          }
                        }
                          else; /* Empty else */
                        }
                      }
                    else;   /* Empty else */
                      }
                    }
                  else; /* Empty else */
                }
                  }
                {
                  {

       /*
        * Communication point BCAST (line 7509)
```

```
* Comm vector: NO COMM VECTOR
* VP-Alive-Var: DPC_temp_38[DPC_vpi]
* Cause of comm:     total_final_exceed = total_final_exceed + 1;
*/

            {
              /* Need block to create scope for DPC_temp_40 and DPC_temp_39 */
              extern int DPC_temp_39 /* Temp for reduce/bcast */ ;
              extern int (*DPC_temp_40) /* Address of temp */ ;

              for (DPC_pkt_count = DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
                {
                  DPC_temp_40 = &DPC_temp_39;

                  /* Check to see if VP is alive */
                  if (DPC_temp_38[DPC_vpi])
                {
                  if (DPC_pkt_count == 0)
                    {
                      /* Do the assignment (only once) */
                      (DPC_temp_39) = (total_final_exceed + 1);

                      /* No packet to fill up for reduce */
                      DPC_pkt_count = 1;
                    }
                  /* No side-effect -- ignore other VPs */
                }
                }   /* End of comm. loop */

              {
                /* Do some nasty stuff for reduce */
                if (DPC_reduce ((void *) &DPC_temp_39,
                        sizeof (DPC_temp_39), (int *) &DPC_temp_40,
                  DPC_pkt_count, DPC_NOP, DPC_INT) == 0)
                  goto DPC_temp_41;

                /* Update the variable that was cause of broadcast */
                (total_final_exceed) = (DPC_temp_39);

              DPC_temp_41:;
                /* If all VPs failed funnel */
              }
              /* End of block for DPC_temp_40 and DPC_temp_39 */
            }

            ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */
              }
            }

            ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */
          }
        }
          }
        }
      }
    }
      }
    }
  }

  ;         /* Just skipped this tree:  <EMPTY EXPRESSION> */
}
  }
```

```
      for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_Cell; DPC_vpi++)
    {
      {
        {
          DPC_temp_31[DPC_vpi] = !((int) (DPC_temp_30[DPC_vpi])) && 1;

          if (DPC_temp_31[DPC_vpi])
        {
          {
            {

              if (DPC_temp_31[DPC_vpi])
            {
              {
                {

                  if (DPC_temp_31[DPC_vpi])
                {
                  {
                    DP[DPC_vpi].data_calc.tree_lai =
                      DP[DPC_vpi].data_calc.shrub_lai =
                      DP[DPC_vpi].data_calc.avg_grass_lai =
                      DP[DPC_vpi].data_calc.total_grass_lai = 0.0;
                    DP[DPC_vpi].data_calc.wood_at =
                      DP[DPC_vpi].data_calc.grass_at =
                      DP[DPC_vpi].data_calc.max_snow =
                      DP[DPC_vpi].data_calc.hi_runoff =
                      DP[DPC_vpi].data_calc.lo_runoff = 0.0;
                    DP[DPC_vpi].data_calc.mo_hi_runoff = -1;
                    DP[DPC_vpi].data_calc.rtop =
                      DP[DPC_vpi].data_calc.tot_runnoff = 0;
                    DP[DPC_vpi].cats.deciduous =
                      DP[DPC_vpi].cats.broadleaf =
                      DP[DPC_vpi].cats.grassfire = 0;
                    DP[DPC_vpi].cats.thaw = -1;
                    DP[DPC_vpi].cats.StationClass = 0;
                  }
                }
                  else; /* Empty else */
                }
              }
            }
              else; /* Empty else */
            }
          }
        }
          else;     /* Empty else */
        }
      }
    }
    }
  }

  _polyprint = 0;

  end_process = time(NULL);
  DoOutput();
  end_write = time(NULL);
}
```

Input Routine

```
/*
**  Dataparallel C Translator, version 0.95
**  Last complete build: 01:10:05 Oct 14 1992
**  Philip Hatcher, Robert Jones, and Anthony Lapadula
**  Copyright (c) 1990, 1991 University of New Hampshire
**  Questions/comments/etc. should be sent to Anthony Lapadula
**      at al@cs.unh.edu or ..!uunet!unhd!al
**  File translated Sat Jul 31 22:27:35 1993
**  Command used for initial pass through preprocessor:
**      "/lib/cpp  -I/usr/include dpc_mapssCPP.dpc"
**  Input file:  dpc_mapssCPP.dpc
**  Output file: dpc_mapssCPP.c
**  Expecting 11 physical processors
*/


void
ReadAscii (in_file,row,col)
  struct _iobuf (*in_file);
  int row;
  int col;
{
  register int DPC_vpi = 0;   /* VP-emulation loop counter */

  int mo;
  int mask;
  char buffer[512];
  char (*ch);
  float ppt;
  float pet;
  float tmp;
  int amask;
  int arow;
  int acol;


  ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */

  if (!dpc_fread(buffer, 264, 1, in_file))
    {
      {
        (void )dpc_printf("Read Failed\n");

        ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */
        exit(18);
      }
    }
  else
    ; /* Empty else */
  mask = atoi(buffer);
  ch = &buffer[12];

  /*
   * Communication point OWNER_ASSIGN (line 5677)
   * Comm vector: NO COMM VECTOR
   * VP-Alive-Var: 1
   * Cause of comm:   ((row) * 201) + col.mask = mask;
   */

  /* No real comm here -- owner of object does assignment */
  if (DPC_nodenum == DPC_get_p_Datum(((row) * 201) + col))
    (DP[DPC_get_vp_Datum((((row) * 201) + col))].mask) = (mask);
```

```
  ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */

  for (mo = 0; mo <= 11; mo++ , ch += 21)
    {
      {
        (void )sscanf(ch, "%f%f%f", &ppt, &pet, &tmp);

        /*
         * Communication point OWNER_ASSIGN (line 5680)
         * Comm vector: NO COMM VECTOR
         * VP-Alive-Var: 1
         * Cause of comm:   ((row) * 201) + col.data_point.ppt[mo] = ppt;
         */

        /* No real comm here -- owner of object does assignment */
        if (DPC_nodenum == DPC_get_p_Datum(((row) * 201) + col))
        (DP[DPC_get_vp_Datum((((row) * 201) + col))].data_point.ppt[mo]) = (ppt);

        ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */

        /*
         * Communication point OWNER_ASSIGN (line 5681)
         * Comm vector: NO COMM VECTOR
         * VP-Alive-Var: 1
         * Cause of comm:   ((row) * 201) + col.data_point.pet[mo] = pet;
         */

        /* No real comm here -- owner of object does assignment */
        if (DPC_nodenum == DPC_get_p_Datum(((row) * 201) + col))
        (DP[DPC_get_vp_Datum((((row) * 201) + col))].data_point.pet[mo]) = (pet);

        ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */

        /*
         * Communication point OWNER_ASSIGN (line 5682)
         * Comm vector: NO COMM VECTOR
         * VP-Alive-Var: 1
         * Cause of comm:   ((row) * 201) + col.data_point.tmp[mo] = tmp;
         */

        /* No real comm here -- owner of object does assignment */
        if (DPC_nodenum == DPC_get_p_Datum(((row) * 201) + col))
        (DP[DPC_get_vp_Datum((((row) * 201) + col))].data_point.tmp[mo]) = (tmp);

        ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */
      }
    }

  ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */
}
```

Output Routine

```
/*
**  Dataparallel C Translator, version 0.95
**  Last complete build: 01:10:05 Oct 14 1992
**  Philip Hatcher, Robert Jones, and Anthony Lapadula
**  Copyright (c) 1990, 1991 University of New Hampshire
**  Questions/comments/etc. should be sent to Anthony Lapadula
**      at al@cs.unh.edu or ..!uunet!unhd!al
**  File translated Sat Jul 31 22:27:35 1993
**  Command used for initial pass through preprocessor:
**      "/lib/cpp  -I/usr/include dpc_mapssCPP.dpc"
**  Input file:  dpc_mapssCPP.dpc
**  Output file: dpc_mapssCPP.c
**  Expecting 11 physical processors
*/


void
DoOutput ()
{
  register int DPC_vpi = 0;   /* VP-emulation loop counter */

  static  int DPC_temp_24 /* OWNER_BCAST result (line 6375) */ ;
  static  int DPC_temp_23 /* OWNER_BCAST result (line 6373) */ ;
  static  int DPC_temp_22 /* OWNER_BCAST result (line 6372) */ ;
  static  int DPC_temp_21 /* OWNER_BCAST result (line 6371) */ ;
  static  int DPC_temp_20 /* OWNER_BCAST result (line 6370) */ ;
  static  int DPC_temp_19 /* OWNER_BCAST result (line 6369) */ ;
  static  float DPC_temp_18   /* OWNER_BCAST result (line 6367) */ ;
  static  int DPC_temp_17 /* OWNER_BCAST result (line 6366) */ ;
  static  float DPC_temp_16   /* OWNER_BCAST result (line 6365) */ ;
  static  int DPC_temp_15 /* OWNER_BCAST result (line 6364) */ ;
  static  int DPC_temp_14 /* OWNER_BCAST result (line 6363) */ ;
  static  float DPC_temp_13   /* OWNER_BCAST result (line 6362) */ ;
  static  float DPC_temp_12   /* OWNER_BCAST result (line 6361) */ ;
  static  float DPC_temp_11   /* OWNER_BCAST result (line 6360) */ ;
  static  float DPC_temp_10   /* OWNER_BCAST result (line 6358) */ ;
  static  float DPC_temp_9  /* OWNER_BCAST result (line 6357) */ ;
  static  float DPC_temp_8  /* OWNER_BCAST result (line 6356) */ ;
  static  float DPC_temp_7  /* OWNER_BCAST result (line 6355) */ ;
  int row;
  int col;


  ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */

  for (row = 0; row < 51; row++)
    {
      {

        for (col = 0; col < 201; col++)
        {
          {

            /*
             * Communication point OWNER_BCAST (line 6355)
             * Comm vector: DPC_temp_7
             * VP-Alive-Var: 1
             * Cause of comm:     ((row) * 201) + col.data_calc.tree_lai;
             */
            {
              int DPC_pp; /* Owner of object to broadcast */
              int DPC_vp; /* Ditto */
```

```

            DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
            DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

            if (DPC_pp == DPC_nodenum)
              {
                /* Copy data into temp for broadcast */
                (DPC_temp_7) = (DP[DPC_vp].data_calc.tree_lai);

              }

            ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
              (void *) &DPC_temp_7;

            ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
              sizeof(DPC_temp_7);
            /* Do the broadcast from this node */
            DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

            /* Every node has up-to-date DPC_temp_7 */

            /* End of block for owner-broadcast */
          }

          /*
           * Communication point OWNER_BCAST (line 6356)
           * Comm vector: DPC_temp_8
           * VP-Alive-Var: 1
           * Cause of comm:     ((row) * 201) + col.data_calc.shrub_lai;
           */
          {
            int DPC_pp; /* Owner of object to broadcast */
            int DPC_vp; /* Ditto */

            DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
            DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

            if (DPC_pp == DPC_nodenum)
              {
                /* Copy data into temp for broadcast */
                (DPC_temp_8) = (DP[DPC_vp].data_calc.shrub_lai);

              }

            ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
              (void *) &DPC_temp_8;

            ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
              sizeof(DPC_temp_8);
            /* Do the broadcast from this node */
            DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

            /* Every node has up-to-date DPC_temp_8 */

            /* End of block for owner-broadcast */
          }

          /*
           * Communication point OWNER_BCAST (line 6357)
           * Comm vector: DPC_temp_9
           * VP-Alive-Var: 1
           * Cause of comm:     ((row) * 201) + col.data_calc.avg_grass_lai;
           */
          {
```

```
      int DPC_pp; /* Owner of object to broadcast */
      int DPC_vp; /* Ditto */

      DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
      DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

      if (DPC_pp == DPC_nodenum)
        {
          /* Copy data into temp for broadcast */
          (DPC_temp_9) = (DP[DPC_vp].data_calc.avg_grass_lai);

        }

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
        (void *) &DPC_temp_9;

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
        sizeof(DPC_temp_9);
      /* Do the broadcast from this node */
      DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

      /* Every node has up-to-date DPC_temp_9 */

      /* End of block for owner-broadcast */
    }

    /*
     * Communication point OWNER_BCAST (line 6358)
     * Comm vector: DPC_temp_10
     * VP-Alive-Var: 1
     * Cause of comm:     ((row) * 201) + col.data_calc.total_grass_lai;
     */
    {
      int DPC_pp; /* Owner of object to broadcast */
      int DPC_vp; /* Ditto */

      DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
      DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

      if (DPC_pp == DPC_nodenum)
        {
          /* Copy data into temp for broadcast */
          (DPC_temp_10) = (DP[DPC_vp].data_calc.total_grass_lai);

        }

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
        (void *) &DPC_temp_10;

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
        sizeof(DPC_temp_10);
      /* Do the broadcast from this node */
      DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

      /* Every node has up-to-date DPC_temp_10 */

      /* End of block for owner-broadcast */
    }

    /*
     * Communication point OWNER_BCAST (line 6360)
     * Comm vector: DPC_temp_11
     * VP-Alive-Var: 1
     * Cause of comm:     ((row) * 201) + col.data_calc.wood_at;
```

```
             */
            {
              int DPC_pp; /* Owner of object to broadcast */
              int DPC_vp; /* Ditto */

              DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
              DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

              if (DPC_pp == DPC_nodenum)
                {
                  /* Copy data into temp for broadcast */
                  (DPC_temp_11) = (DP[DPC_vp].data_calc.wood_at);

                }

              ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
                (void *) &DPC_temp_11;

              ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
                sizeof(DPC_temp_11);
              /* Do the broadcast from this node */
              DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

              /* Every node has up-to-date DPC_temp_11 */

              /* End of block for owner-broadcast */
            }

            /*
             * Communication point OWNER_BCAST (line 6361)
             * Comm vector: DPC_temp_12
             * VP-Alive-Var: 1
             * Cause of comm:     ((row) * 201) + col.data_calc.grass_at;
             */
            {
              int DPC_pp; /* Owner of object to broadcast */
              int DPC_vp; /* Ditto */

              DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
              DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

              if (DPC_pp == DPC_nodenum)
                {
                  /* Copy data into temp for broadcast */
                  (DPC_temp_12) = (DP[DPC_vp].data_calc.grass_at);

                }

              ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
                (void *) &DPC_temp_12;

              ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
                sizeof(DPC_temp_12);
              /* Do the broadcast from this node */
              DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

              /* Every node has up-to-date DPC_temp_12 */

              /* End of block for owner-broadcast */
            }

            /*
             * Communication point OWNER_BCAST (line 6362)
             * Comm vector: DPC_temp_13
```

```
 * VP-Alive-Var: 1
 * Cause of comm:     ((row) * 201) + col.data_calc.max_snow;
 */
{
  int DPC_pp; /* Owner of object to broadcast */
  int DPC_vp; /* Ditto */

  DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
  DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

  if (DPC_pp == DPC_nodenum)
    {
      /* Copy data into temp for broadcast */
      (DPC_temp_13) = (DP[DPC_vp].data_calc.max_snow);

    }

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
    (void *) &DPC_temp_13;

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
    sizeof(DPC_temp_13);
  /* Do the broadcast from this node */
  DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

  /* Every node has up-to-date DPC_temp_13 */

  /* End of block for owner-broadcast */
}

/*
 * Communication point OWNER_BCAST (line 6363)
 * Comm vector: DPC_temp_14
 * VP-Alive-Var: 1
 * Cause of comm:     ((row) * 201) + col.data_calc.rtop;
 */
{
  int DPC_pp; /* Owner of object to broadcast */
  int DPC_vp; /* Ditto */

  DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
  DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

  if (DPC_pp == DPC_nodenum)
    {
      /* Copy data into temp for broadcast */
      (DPC_temp_14) = (DP[DPC_vp].data_calc.rtop);

    }

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
    (void *) &DPC_temp_14;

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
    sizeof(DPC_temp_14);
  /* Do the broadcast from this node */
  DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

  /* Every node has up-to-date DPC_temp_14 */

  /* End of block for owner-broadcast */
}

/*
```

```
 * Communication point OWNER_BCAST (line 6364)
 * Comm vector: DPC_temp_15
 * VP-Alive-Var: 1
 * Cause of comm:     ((row) * 201) + col.data_calc.tot_runnoff;
 */
{
  int DPC_pp; /* Owner of object to broadcast */
  int DPC_vp; /* Ditto */

  DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
  DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

  if (DPC_pp == DPC_nodenum)
    {
      /* Copy data into temp for broadcast */
      (DPC_temp_15) = (DP[DPC_vp].data_calc.tot_runnoff);

    }

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
    (void *) &DPC_temp_15;

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
    sizeof(DPC_temp_15);
  /* Do the broadcast from this node */
  DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

  /* Every node has up-to-date DPC_temp_15 */

  /* End of block for owner-broadcast */
}

/*
 * Communication point OWNER_BCAST (line 6365)
 * Comm vector: DPC_temp_16
 * VP-Alive-Var: 1
 * Cause of comm:     ((row) * 201) + col.data_calc.hi_runoff;
 */
{
  int DPC_pp; /* Owner of object to broadcast */
  int DPC_vp; /* Ditto */

  DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
  DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

  if (DPC_pp == DPC_nodenum)
    {
      /* Copy data into temp for broadcast */
      (DPC_temp_16) = (DP[DPC_vp].data_calc.hi_runoff);

    }

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
    (void *) &DPC_temp_16;

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
    sizeof(DPC_temp_16);
  /* Do the broadcast from this node */
  DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

  /* Every node has up-to-date DPC_temp_16 */

  /* End of block for owner-broadcast */
}
```

```
        /*
         * Communication point OWNER_BCAST (line 6366)
         * Comm vector: DPC_temp_17
         * VP-Alive-Var: 1
         * Cause of comm:     ((row) * 201) + col.data_calc.mo_hi_runoff;
         */
        {
          int DPC_pp; /* Owner of object to broadcast */
          int DPC_vp; /* Ditto */

          DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
          DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

          if (DPC_pp == DPC_nodenum)
            {
              /* Copy data into temp for broadcast */
              (DPC_temp_17) = (DP[DPC_vp].data_calc.mo_hi_runoff);

            }

          ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
            (void *) &DPC_temp_17;

          ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
            sizeof(DPC_temp_17);
          /* Do the broadcast from this node */
          DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

          /* Every node has up-to-date DPC_temp_17 */

          /* End of block for owner-broadcast */
        }

        /*
         * Communication point OWNER_BCAST (line 6367)
         * Comm vector: DPC_temp_18
         * VP-Alive-Var: 1
         * Cause of comm:     ((row) * 201) + col.data_calc.lo_runoff;
         */
        {
          int DPC_pp; /* Owner of object to broadcast */
          int DPC_vp; /* Ditto */

          DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
          DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

          if (DPC_pp == DPC_nodenum)
            {
              /* Copy data into temp for broadcast */
              (DPC_temp_18) = (DP[DPC_vp].data_calc.lo_runoff);

            }

          ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
            (void *) &DPC_temp_18;

          ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
            sizeof(DPC_temp_18);
          /* Do the broadcast from this node */
          DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

          /* Every node has up-to-date DPC_temp_18 */

```

```
    /* End of block for owner-broadcast */
  }

  /*
   * Communication point OWNER_BCAST (line 6369)
   * Comm vector: DPC_temp_19
   * VP-Alive-Var: 1
   * Cause of comm:     ((row) * 201) + col.cats.thaw;
   */
  {
    int DPC_pp; /* Owner of object to broadcast */
    int DPC_vp; /* Ditto */

    DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
    DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

    if (DPC_pp == DPC_nodenum)
      {
        /* Copy data into temp for broadcast */
        (DPC_temp_19) = (DP[DPC_vp].cats.thaw);

      }

    ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
      (void *) &DPC_temp_19;

    ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
      sizeof(DPC_temp_19);
    /* Do the broadcast from this node */
    DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

    /* Every node has up-to-date DPC_temp_19 */

    /* End of block for owner-broadcast */
  }

  /*
   * Communication point OWNER_BCAST (line 6370)
   * Comm vector: DPC_temp_20
   * VP-Alive-Var: 1
   * Cause of comm:     ((row) * 201) + col.cats.deciduous;
   */
  {
    int DPC_pp; /* Owner of object to broadcast */
    int DPC_vp; /* Ditto */

    DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
    DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

    if (DPC_pp == DPC_nodenum)
      {
        /* Copy data into temp for broadcast */
        (DPC_temp_20) = (DP[DPC_vp].cats.deciduous);

      }

    ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
      (void *) &DPC_temp_20;

    ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
      sizeof(DPC_temp_20);
    /* Do the broadcast from this node */
    DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

```

```
      /* Every node has up-to-date DPC_temp_20 */

      /* End of block for owner-broadcast */
    }

    /*
     * Communication point OWNER_BCAST (line 6371)
     * Comm vector: DPC_temp_21
     * VP-Alive-Var: 1
     * Cause of comm:     ((row) * 201) + col.cats.broadleaf;
     */
    {
      int DPC_pp; /* Owner of object to broadcast */
      int DPC_vp; /* Ditto */

      DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
      DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

      if (DPC_pp == DPC_nodenum)
        {
          /* Copy data into temp for broadcast */
          (DPC_temp_21) = (DP[DPC_vp].cats.broadleaf);

        }

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
        (void *) &DPC_temp_21;

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
        sizeof(DPC_temp_21);
      /* Do the broadcast from this node */
      DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

      /* Every node has up-to-date DPC_temp_21 */

      /* End of block for owner-broadcast */
    }

    /*
     * Communication point OWNER_BCAST (line 6372)
     * Comm vector: DPC_temp_22
     * VP-Alive-Var: 1
     * Cause of comm:     ((row) * 201) + col.cats.grassfire;
     */
    {
      int DPC_pp; /* Owner of object to broadcast */
      int DPC_vp; /* Ditto */

      DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
      DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

      if (DPC_pp == DPC_nodenum)
        {
          /* Copy data into temp for broadcast */
          (DPC_temp_22) = (DP[DPC_vp].cats.grassfire);

        }

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
        (void *) &DPC_temp_22;

      ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
        sizeof(DPC_temp_22);
      /* Do the broadcast from this node */
```

```
  DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

  /* Every node has up-to-date DPC_temp_22 */

  /* End of block for owner-broadcast */
}

/*
 * Communication point OWNER_BCAST (line 6373)
 * Comm vector: DPC_temp_23
 * VP-Alive-Var: 1
 * Cause of comm:     ((row) * 201) + col.cats.StationClass;
 */
{
  int DPC_pp; /* Owner of object to broadcast */
  int DPC_vp; /* Ditto */

  DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
  DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

  if (DPC_pp == DPC_nodenum)
    {
      /* Copy data into temp for broadcast */
      (DPC_temp_23) = (DP[DPC_vp].cats.StationClass);

    }

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
    (void *) &DPC_temp_23;

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
    sizeof(DPC_temp_23);
  /* Do the broadcast from this node */
  DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

  /* Every node has up-to-date DPC_temp_23 */

  /* End of block for owner-broadcast */
}

/*
 * Communication point OWNER_BCAST (line 6375)
 * Comm vector: DPC_temp_24
 * VP-Alive-Var: 1
 * Cause of comm:     ((row) * 201) + col.mask;
 */
{
  int DPC_pp; /* Owner of object to broadcast */
  int DPC_vp; /* Ditto */

  DPC_pp = DPC_get_p_Datum(((row) * 201) + col    );
  DPC_vp = DPC_get_vp_Datum(((row) * 201) + col    );

  if (DPC_pp == DPC_nodenum)
    {
      /* Copy data into temp for broadcast */
      (DPC_temp_24) = (DP[DPC_vp].mask);

    }

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].src_addr =
    (void *) &DPC_temp_24;

  ((struct DPC_bcast_pkt *) DPC_pkts)[0].obj_len =
```

```
                sizeof(DPC_temp_24);
              /* Do the broadcast from this node */
              DPC_broadcast (DPC_pp, DPC_pkts, (DPC_pp == DPC_nodenum));

              /* Every node has up-to-date DPC_temp_24 */

              /* End of block for owner-broadcast */
            }
            (void )dpc_printf(
                "dpc_%dX%d %8.3f %8.3f %8.3f %8.3f %d %d %d %d %d %d %d %d %d %d %d %d %d %d\n",
                row, col,
                DPC_temp_7, DPC_temp_8, DPC_temp_9, DPC_temp_10,
                (int )(((double )(DPC_temp_11)) + 0.5),
                (int )(((double )(DPC_temp_12)) + 0.5),
                (int )(((double )(DPC_temp_13)) + 0.5),
                DPC_temp_14, DPC_temp_15,
                (int )(((double )(DPC_temp_16)) + 0.5),
                DPC_temp_17, (int )(((double )(DPC_temp_18)) + 0.5),
                DPC_temp_19, DPC_temp_20, DPC_temp_21,
                DPC_temp_22, DPC_temp_23, DPC_temp_24);
          }
        }
      }
    }

  ;   /* Just skipped this tree:  <EMPTY EXPRESSION> */
}
```