April 1993
Order Number: 312545-001

PARAGON™ OSF/1 SOFTWARE TOOLS

USER'S GUIDE

Intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	4/93

LIMITED RIGHTS

Preface

This manual describes the software development tools, applications, and application libraries that run on the Paragon™ OSF/1 operating system. As additional tools become available, chapters describing these new tools and libraries will be included in future releases of this book.

Organization

Chapter 1 — Describes the use of the X Window System client libraries with the Paragon™ OSF/1 operating system, including special programming techniques for node X programs and how to compile and link X applications.

Chapter 2 — Describes the use of the DGL client libraries, including how to compile and link DGL applications, and special programming techniques.

Chapter 3 — Describes the **pmake** parallel make utility.

Notational Conventions

This manual uses the following notational conventions:

Bold — Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic — Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace` — Identifies computer output (prompts and messages), examples, and values of variables.

`Bold-Italic-Monospace` — Identifies user input (what you enter in response to some prompt).

Bold-Monospace
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

<Break> **<s>** **<Ctrl-Alt-Del>**

[ ] (Brackets) Surround optional items.

... (Ellipsis) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items, of which you may select only one.

{ } (Braces) Surround two or more items, of which you must select one.

Applicable Documents

For more information, refer to the following manuals.

Paragon™ OSF/1 Manuals

- *Paragon™ OSF/1 User's Guide*
- *Paragon™ OSF/1 Commands Reference Manual*
- *Paragon™ OSF/1 C System Calls Reference Manual*
- *Paragon™ OSF/1 Fortran Compiler User's Guide*

Other Manuals

- *OSF/1 User's Guide*
- *Xlib Programming Manual* (O'Reilly and Associates)
- *X Protocol Reference Manual* (O'Reilly and Associates)
- *X Toolkit Intrinsics Programming Manual* (O'Reilly and Associates)
- *X Toolkit Intrinsics Reference Manual* (O'Reilly and Associates)
- *Graphics Library Programming Guide* - Silicon Graphics, Inc. (Available from SGI; SGI order number 007-1210-040)
- *The GNU Make Manual*

Comments and Assistance

Intel Supercomputer Systems Division is eager to hear of your experiences with our new software product. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
phone: 800-421-2823
email: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Japan Intel Corporation K.K.
Supercomputer Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Supercomputer System Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bel Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Supercomputer Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 629-7600

Table of Contents

Chapter 1
The X Window System on the Nodes

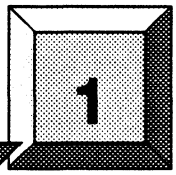
Chapter 2
Using the Distributed Graphics Library

Chapter 3
The Parallel Make Utility

List of Illustrations

List of Tables

The X Window System on the Nodes



Introduction

The X Window System, developed during Project Athena at the Massachusetts Institute of Technology, is a software industry standard for graphics programming. It provides a standard environment for application software and can control workstation displays.

A set of X Window System client libraries is included with Paragon™ OSF/1. Applications using the X Window System must be written in the C language.

This chapter describes:

- Special programming techniques for node X programs.
- How to compile and link X applications.
- What to do if your node X program cannot open the display server.

This chapter contain information specific to writing X applications for Paragon OSF/1 only. It does not describe how to write X Window System applications programs. For general information on writing X programs, refer to the X Window System manuals by O'Reilly and Associates.

To use your workstation as a server that accesses the client libraries, the X server software must be installed on your workstation. Most versions of the X server software have an authorization mechanism to limit access of clients on other nodes of the network to your display. For more information on authorization and security, refer to the X server documentation for your workstation, the X online manual page, and "Authorizing the Supercomputer to Access the Server" on page 1-19.

The TCP/IP software on your Intel supercomputer must also be properly configured to install and use the X software, and an entry for your X server must be included in the Intel supercomputer's */etc/hosts* file or NIS database. If no such entry exists, your system administrator must add an entry for your server. For more information, refer to "Ensuring that Supercomputer and Server Know Each Other's Address" on page 1-18.

Most of the programming techniques described in this chapter apply to programs to be run in the compute partition. If you create an X program to run only in the service partition, you need only link service partition applications properly, as described in "Compiling and Linking X Window System Applications" on page 1-15. No special programming techniques are necessary unless your program uses multiple threads; if it does, see "Threads and Mutexes" on page 1-7.

A Sample Node X Program

To help you start using X in the compute partition, a sample program called *graph* is in the directory */user/share/examples/c/xtoolkit* on your Paragon system. This program demonstrates the special programming techniques that you can use to write X applications to run on multiple nodes in the compute partition. A *makefile* is available in the same directory. Because the program is too long to print in its entirety, this chapter explains only selected parts.

If you create an X program to run only in the service partition, no special considerations are necessary. You need only see the system link instructions.

Compiling and running the *graph* program can help you verify that your Intel supercomputer and server are properly configured.You might also wish to use it as a basis for your own X program, or you may wish to examine the code for programming techniques.

The *graph* program uses the X Toolkit and the Athena Widgets. Toolkit programs are easier to write and maintain than *Xlib* programs, and can offer more functionality with very little additional effort. This chapter does not discuss the basic concepts of Toolkit and widget programming; for information on these topics, refer to Volumes 4 and 5 of the O'Reilly and Associates manuals.

What the *graph* Program Does

The *graph* program performs the simple calculation for the value of **sin**(x) for $0 <= x < 2\pi$ and graphs the result in a window as it calculates. The problem decomposition used in this example is a modified domain decomposition. Node 0 maintains the display, and the other nodes calculate the points of the curve. The x values from 0 to 2π are divided evenly among the calculating nodes. For example, if the program is run on four nodes, node 1 is responsible for $0 <= x <= 2\pi/3$, node 2 is responsible for $2\pi/3 < x <= 4\pi/3$, and node 3 is responsible for $4\pi/3 < x < 2\pi$.

Figure 1-1 shows the window and what the *graph* program displays when it is running. You can resize the window using your window manager. If the window becomes too small to display the graph, horizontal and vertical scroll bars appear. The program adds two buttons to the window to allow additional control during calculation. A **restart** button clears the window and restarts the calculation from the beginning. A **quit** button allows you to terminate the program.In addition, your window manager may also place a title bar and/or border on the window, which are not shown.

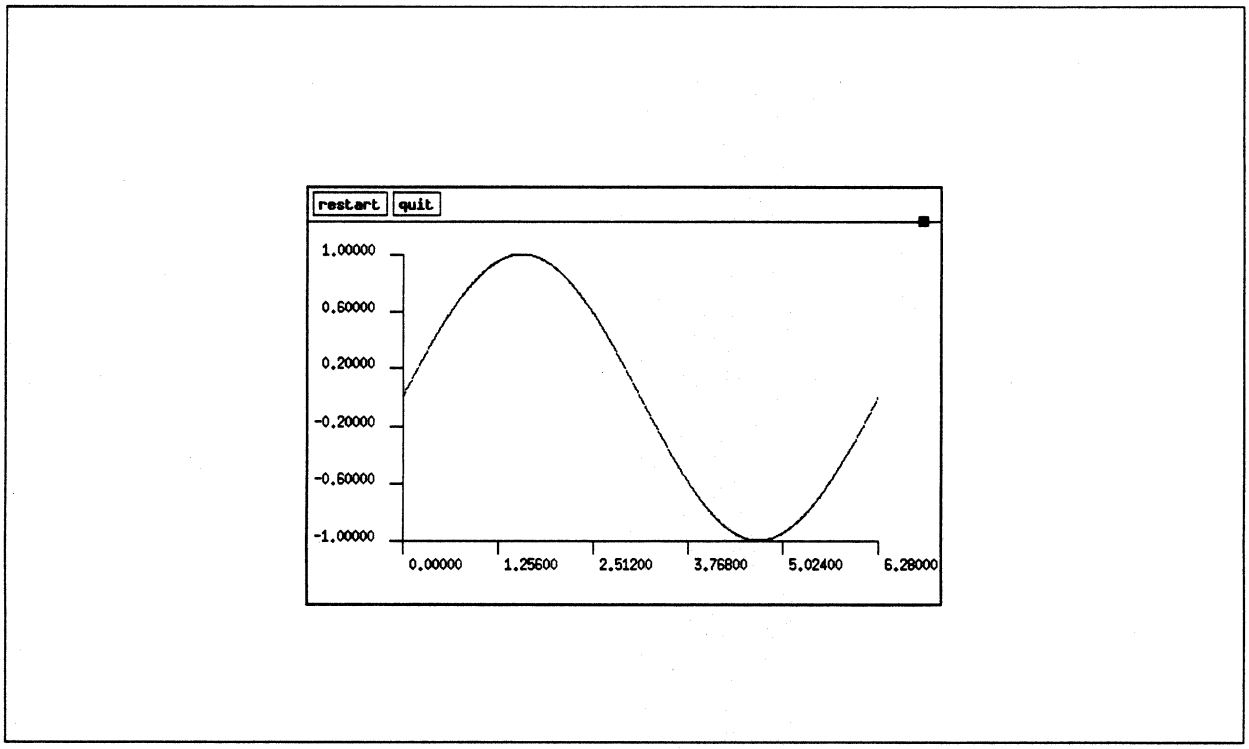


Figure 1-1. The *graph* Program Display

Compiling, Linking, and Executing the *graph* Program

This section gives you step-by-step instructions for compiling and linking the *graph* program. To compile this program on your workstation, you must have the **icc** cross-compiler.

1. Create a directory for the *graph* program in your current directory with the following command:

   ```
   mkdir graph
   ```

2. Copy the source code and *makefile* for the *graph* program to your *graph* directory.

   On the Intel supercomputer, use the following command:

   ```
   cp /user/share/examples/c/toolkit/graph/* graph
   ```

On a workstation, use **rcp**, **ftp**, or NFS to transfer the file from the supercomputer to your workstation. For example, you could use **rcp**, as in the following command, replacing *super* with the name of your Intel supercomputer, and *path* with the path of the example:

```
rcp "super:/user/share/examples/c/xtoolkit/graph/*" graph
```

3. Change to the *graph* program directory:

```
cd graph
```

4. Use the supplied *makefile* to compile and link the program by entering the following:

```
make
```

5. If you compiled the program on a workstation, copy the executable to the Intel supercomputer and then log into the Intel supercomputer, using the appropriate command for your site. For example, if the appropriate commands are **rcp** and **rlogin**, use the following commands:

```
rcp graph super:
rlogin super
```

6. Set the *DISPLAY* environment variable to the appropriate value for your server. For example, if your shell is **csh** and your server is a workstation called *mysun*, use the following command:

```
setenv DISPLAY mysun:0
```

7. Verify that your supercomputer is authorized to connect to your X Window System server. For example, if you are using **xhost**-based authentication, you would execute the following command on your workstation:

```
xhost super
```

8. Run the *graph* program on at least two nodes; four or more are recommended. For example, to run the program on eight nodes of your default partition, use the following command:

```
graph -sz 8
```

The *graph* window appears and the graph is drawn. If the message "Error: Can't Open display" appears instead, refer to "Problems in Opening the Display" on page 1-17.

Refer to the *Paragon™ OSF/1 User's Guide* for information on controlling the execution of parallel applications.

9. To draw the graph again, click the button labeled **restart**. To quit, click the **quit** button.

Widget Hierarchy of the *graph* Program

The *graph* program uses six types of widgets, not all of which are visible in Figure 1-1. Figure 1-2 shows how these widgets relate to each other in a widget hierarchy. Higher-level widgets, closer to the back, are called "*parents*; lower-level widgets, closer to the front, are called *children*. Parent widgets manage their children; child widgets provide the basic functionality of the program.

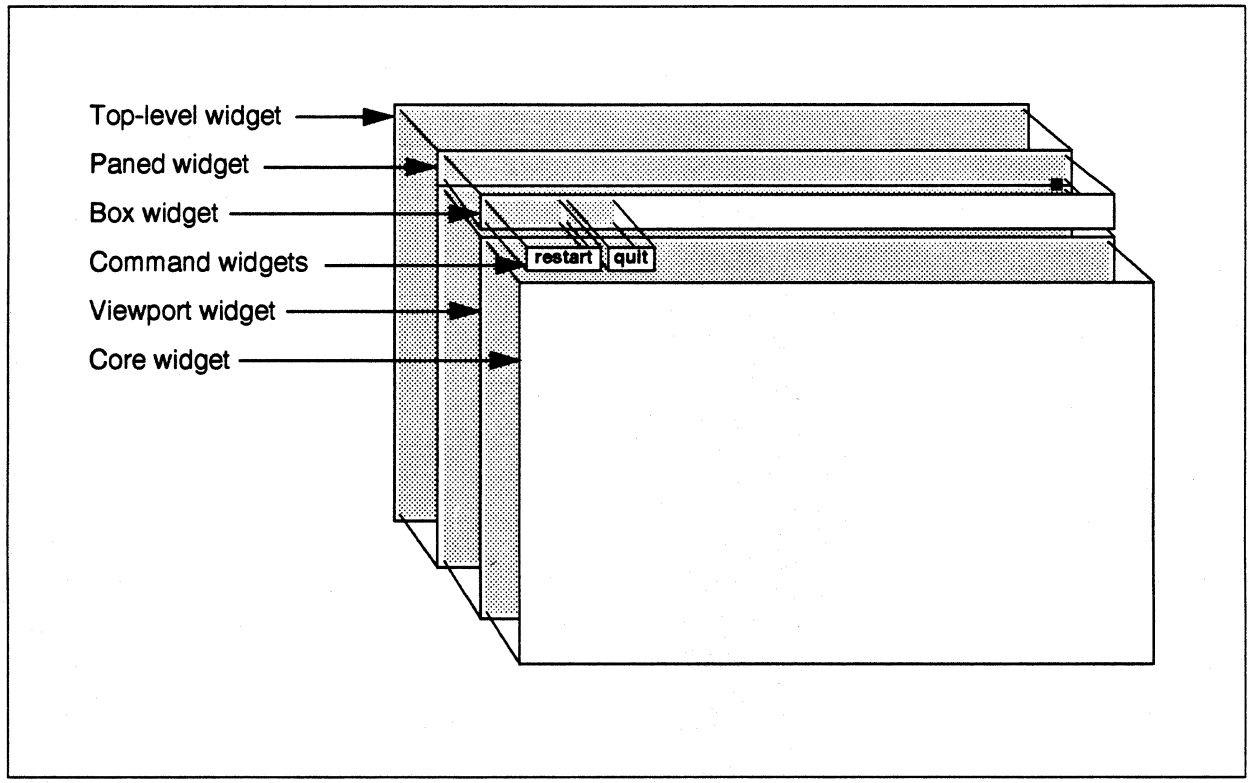


Figure 1-2. *graph* Program Widget Hierarchy

The following list describes the widgets used in the program. Widgets of these types are commonly used in X programs.

- An invisible *top-level widget* provides an interface between the window manager and its children. When you resize a window using the window manager, the top-level widget transmits the changes to its child widgets. This top-level widget has one child: a Paned widget.
- A *Paned widget* holds its children in a series of vertical areas called "panes." When the Paned widget is resized, it resizes its children to display as much information as possible in the new size. This Paned widget has two children, a Box widget and a Viewport widget.
- A *Box widget* holds its children in an arbitrary arrangement. The Box widget in the *graph* program has two children, both Command widgets, arranged side-by-side.
- A *Viewport widget* provides scroll bars when necessary, and scrolls its children when you click the mouse in the scroll bars. This Viewport widget has one child: a Core widget.
- A *Command widget* is a pushbutton that invokes a function when the user clicks on it. The *graph* program has two Command widgets that invoke the **Restart** and **Quit** functions.
- A *Core widget*, a kind of "vanilla" widget, does not have any functionality of its own. The *graph* program uses the Core widget as a drawing surface for the graph.

In addition to its widgets, the program maintains an off-screen bitmap (a pixmap of depth 1) as a graphics buffer. When the program draws the graph, it does not draw directly into the window. Instead, it draws into its bitmap and then copies the bitmap into the Core widget. Thus, if the window is obscured and then exposed, the bitmap can be copied to the widget again, without having to recalculate the points of the graph.

Programming Techniques

This section discusses the special programming techniques for writing X applications that run on multiple nodes. These techniques are demonstrated in the *graph* program.

Node/Server Connection

For a node to have a connection to the server, the node must open its own connection with a call to **XOpenDisplay**(). This can either be a direct call, or through some higher-level function such as **XtAppInitialize**()).

Because each call to **XOpenDisplay**() uses a file descriptor in the server, it is usually better to have one node make all of the X system calls (the distinguished node method). This node opens a connection to the display and handles the display. Most X Window System servers have a limited number of file descriptors, which limits the number of nodes that can open simultaneous connections to the server. Increasing the file descriptor limit would require configuring a new system kernel and rebuilding the X Window System server for your workstation. For more information, refer to your workstation documentation or your system administrator.

In the *graph* program, only node 0 makes X calls. The other nodes only calculate; they pass the results of their calculations to node 0 as messages. Node 0 graphs the contents of each message it receives. In the *graph* program, node 0 does not calculate; it only handles the display. This design is simpler to program, but requires the use of two or more nodes. You could, in another program, decide to have this node calculate as well. The choice depends upon the relative importance of speedy calculation and good graphics performance for your application.

Combining X Event-Driven Programming with Message Passing

X programs are built around a "main event loop," which repeatedly retrieves and manipulates an X event. The problem with using a loop like this in a message-passing node program is that the program blocks while waiting for the next event, because **XNextEvent**() does not return until there is an event. Thus, when no X events are coming in, the node is blocked and cannot calculate or deal with messages from other nodes. To avoid this problem, you can use *threads*.

Threads and Mutexes

In Paragon OSF/1, a process consists of a set of *resources* such as memory objects and open files, and one or more *threads*. A *thread*, short for *thread of control*, is also called a *lightweight process*. A thread consists of an instruction pointer and a stack. For example, a process created by a call to **fork**() contains one thread. The standard OSF/1 library call **pthread_create**() creates additional threads.

When more than one thread is in a single process, each thread executes independently, but they share resources. For example, all the threads in a single process share memory; when one thread writes to a variable in memory, it modifies the value of that variable for all threads. All the threads in a process run on the same node.

Because threads share memory, you must carefully coordinate access to shared areas of memory. For example, if two threads write to the same area of memory at the same time, the results may be indeterminate. To prevent this problem, you can use a *mutex* (short for *multiple exclusion*) variable.

A mutex is a variable of type **pthread_mutex_t** (defined in *pthread.h*). A mutex has two states: locked and unlocked. The call **pthread_mutex_init()** initializes a mutex and should be called before the **pthread_create()** so that the two threads can share the mutex. The call **pthread_mutex_lock()** attempts to lock a mutex. If it is already locked, the call blocks until the mutex becomes unlocked. The call **pthread_mutex_unlock()** unlocks a mutex. By surrounding writes to shared variables with **pthread_mutex_lock()** and **pthread_mutex_unlock()**, you can make sure that only one thread writes to the variable at the same time.

NOTE

In programs where multiple threads make X calls, you must surround all X calls in all threads with **pthread_mutex_lock()** and **pthread_mutex_unlock()**.

This is required because the X library is not reentrant; that is, it is not written to be executed by multiple threads simultaneously. If two threads make calls to the X library simultaneously, even if they are different calls, internal static variables used by the X library could take on indeterminate values. This could result in hard-to-diagnose X errors.

Threads and mutexes are a standard part of OSF/1 and are not unique to Paragon OSF/1. For more information on threads and mutexes, refer to **pthread_create()** and **pthread_mutex_init()** in the *OSF/1 Programmer's Reference.*

Threads and Mutexes in the *graph* Program

The *graph* program begins by having node 0 set up all the widgets, windows, and other X structures it will need. As part of this step, it initializes a mutex with the default attributes:

```
#include <pthread.h>
pthread_mutex_t mutex;  /* mutual-exclusion semaphore */
        .
        .
        .
    /* create a mutex to control access to the X server */
    pthread_mutex_init(&mutex, pthread_mutexattr_default);
```

The variable *mutex* is declared as a global variable so that all the functions in the program can use the same mutex. The special value **pthread_mutexattr_default** is defined in *pthread.h* and specifies the default attributes for a mutex.

After setting up all the X objects, the *graph* program creates a thread to handle X events:

```
    pthread_t thread;
        .
        .
        .
    /* create a thread to handle X events */
    pthread_create(&thread, pthread_attr_default,
                   (void *(*)())XtAppMainLoop,
                   (void *)app_context);
```

The first two arguments of **pthread_create()** specify the thread and its attributes, as is the case for **pthread_mutex_init()**. The third and fourth arguments specify the function that the thread should begin executing and the arguments of that function. In this example, it calls the standard X toolkit function **XtAppMainLoop()** with the argument *app_context*. This function never returns; it loops forever, handling any X events that arise.

While the thread created by this call to **pthread_create()** is handling the X events, the program's original thread of control also enters an infinite loop. This loop repeatedly waits for a message from another node that contains a series of data points, draws those points into an off-screen bitmap, and then copies the bitmap to the screen by calling **RedrawPicture()**:

```
/* infinite loop for messages */
while(1) {
    crecv(DATA, &points, sizeof(points));

    pthread_mutex_lock(&mutex);
    XDrawPoints(XtDisplay(bitmap), picture, draw_gc,
                points.points, points.npoints,
                CoordModeOrigin);
    pthread_mutex_unlock(&mutex);

    RedrawPicture(bitmap, NULL, NULL, NULL);
}
```

The **XDrawPoints()** call is surrounded by **pthread_mutex_lock()** and **pthread_mutex_unlock()** calls on the mutex named *mutex*. The same pair of calls surrounds each X call in the program, except for those that precede the **pthread_create()**. This prevents either thread from making any X call if the other thread is in the process of making an X call.

The function **RedrawPicture()** is not surrounded by lock and unlock calls because it is a user-provided function, not a standard X Toolkit function. However, X calls within **RedrawPicture()** *are* surrounded by lock and unlock calls. The code for **RedrawPicture()** is shown in "Starting the Other Nodes on the First Expose Event" on page 1-11.

Avoid locking a mutex before calling a user-provided function that locks the same mutex. If the mutex is locked before the call, the lock within the call waits for the mutex to unlock. Because the thread is waiting for itself, the mutex will never unlock, hanging the process.

Synchronizing Window Operations with Window Mapping

X programs should wait until the window has actually appeared before manipulating it. The function **XMapWindow()** and similar functions do not necessarily cause the window to appear immediately; they merely request that the window be mapped. Depending on the window manager, the window may not appear for a while.

You can have the program wait until the first **Expose** event is received from the window, or you can insert a call to **XSync()** after the **XMapWindow()**. Attempting to perform certain operations on a window that has not yet appeared can result in X errors.

In node programs, ensuring that the window has appeared before any node attempts to use it can be a special problem. It would be simplest to have the X node (the node with the connection to the display) immediately display the results of each message received. If, however, the calculating nodes, which may or may not include the X node, begin sending data to the X node as soon as they start up, the X node may receive data to draw before the window is available. You can deal with this problem in several ways, such as:

- Have the calculating nodes begin calculating as soon as they are loaded, sending the results to the X node as soon as they are available. The X node buffers the data from the calculating nodes as the data is received, then draws the contents of the window all at once after the first event is received. This requires more buffering on the X node.

- Have the calculating nodes wait to calculate until they receive a "go-ahead" message from the X node. The X node sends this message when it receives the first event. The "go-ahead" message may include data the node needs to begin calculation. This technique can be used in a manager-worker problem decomposition to make the X node the manager.

- Have the calculating nodes begin calculating as soon as they are loaded, buffering any results generated. The X node sends a "go-ahead" message to the calculating nodes when the first event is received, and the other nodes send a return message with the buffered data when they receive the "go-ahead." This technique requires more buffering on the calculating nodes.

In the *graph* program, the computing nodes begin calculating only when the X node tells them to start, sending a "go-ahead" message when the first **Expose** event is received.

Starting the Other Nodes on the First Expose Event

The **RedrawPicture**() function in the *graph* program is called in response to an **Expose** event. Following is the **RedrawPicture**() function code:

```
static void
RedrawPicture(w, event, params, num_params)
Widget w;
XExposeEvent *event;     /* ignored */
String *params;          /* ignored */
Cardinal *num_params;    /* ignored */
{
    static int started = 0;

    pthread_mutex_lock(&mutex);
    if (DefaultDepthOfScreen(XtScreen(w)) == 1) {
        XCopyArea(XtDisplay(bitmap), picture, XtWindow(bitmap),
                  copy_gc, 0, 0, PIXMAPWIDTH, PIXMAPHEIGHT,
                  0, 0);
    } else {
        XCopyPlane(XtDisplay(bitmap), picture,
                   XtWindow(bitmap),copy_gc, 0, 0,
                   PIXMAPWIDTH, PIXMAPHEIGHT, 0, 0, 1);
    }
    pthread_mutex_unlock(&mutex);

    if(!started) {
        StartNodes();
        started = 1;
    }
}
```

The **RedrawPicture()** function is an *action procedure*, a function that is called by a widget in response to an event. All action procedures have the same four parameters as shown. In this example, this action procedure uses only the first parameter, and performs two tasks:

1. Copies the program's off-screen bitmap to the Core widget, using **XCopyArea()** if the screen is monochrome and **XCopyPlane()** if the screen is color or grayscale. **XCopyPlane()** is slower than **XCopyArea()**, but is necessary when a monochrome bitmap is copied to a color or grayscale screen.

   The entire **if** test is surrounded by **pthread_mutex_lock()** and **pthread_mutex_unlock()** calls to prevent the two threads in the program from making any X calls at the same time. The **if** test contains the X calls **DefaultDepthOfScreen()**, **XCopyArea()**, and **XCopyPlane()**.

2. On the first call, the function sends a "go-ahead" message to the other nodes to indicate that the window has been exposed and the other nodes can start sending data. This message prevents the program from trying to draw in the window before the window exists.

The function **StartNodes()**, called in the **RedrawPicture()** function, sends an empty message of type **START** (an arbitrary constant defined earlier in the program) to the other nodes:

```
static void
StartNodes()
{
    csend(START, NULL, 0, -1, 0);
}
```

In a real application, this would be a good place to send each node its initial data.

Associating a Function with an Expose Event

The following code creates the widget and makes it call the function **RedrawPicture()** when it receives an **Expose** event.

```
/* translation table for bitmap core widget */
String trans = "<Expose>:    RedrawPicture()";
    •
    •
    •
/* create Core widget for drawing into */
bitmap = XtVaCreateManagedWidget("bitmap",
            widgetClass, viewport,
            XtNtranslations, XtParseTranslationTable(trans),
            XtNwidth, PIXMAPWIDTH,
            XtNheight, PIXMAPHEIGHT,
            NULL);
```

This code is fairly complicated, because a widget of class **widgetClass** (a Core widget) does not have any pre-defined action procedures. You have to create a string called a "translation table" that associates event types with function names, and then use **XtParseTranslationTable()** to convert the translation table to a form that the Toolkit can interpret. The translation table in this example contains only one line, which makes the widget call the function **RedrawPicture()** whenever it receives an **Expose** event.

Within this section of code, the following lines in the call to **XtVaCreateManagedWidget()** set the widget's width resource to **PIXMAPWIDTH** and its height resource to **PIXMAPHEIGHT**:

```
            XtNwidth, PIXMAPWIDTH,
            XtNheight, PIXMAPHEIGHT,
```

PIXMAPWIDTH and **PIXMAPHEIGHT** are the size of the pixmap, defined earlier in the program. These lines are necessary because the default size of the Core widget is 0 by 0 pixels.

Responding to Window Destruction

If the user destroys the application window using the window manager's Kill Window function, the default window destruction procedures terminate the X node, but do *not* terminate the calculating nodes.

To ensure that the destruction of the application window does not become a problem, you can install an I/O error handler that kills the other node processes by calling **kill(0, SIGKILL)**. The proper way to install this error handler depends upon the toolkit (if any) and the window manager you are using, There is no error handler in the *graph* program. For more information on handling window destruction, refer to the documentation for your toolkit and window manager.

Batching Data Points into Larger Messages for Improved Performance

Graphics performance usually suffers when you draw only one point at a time. The performance of a message-passing program also suffers when you spend a lot of time passing messages. You can improve both kinds of performance by batching your data points into larger messages.

The following is the code that collects the data into messages in the *graph* program:

```
struct {
    int npoints;
    XPoint points[NPOINTS];
} points;       /* data points */
        .
        .
        .
    double x, y, unit, start, end;
    int i = 0;
        .
        .
        .
            for(x = start; x < end; x += STEP) {
                y = sin(x);
                ProbToScreen(x, y, &(points.points[i]));
                i++;
                if(i >= NPOINTS) {
                    points.npoints = i;
                    csend(DATA, &points, sizeof(points),
                          0, 0);
                    i = 0;
                }
            }

            if(i != 0) {
```

```
        points.npoints = i;
        csend(DATA, &points, sizeof(points), 0, 0);
    }
```

The user-provided function **ProbToScreen()** transforms problem coordinates (the units used in the calculation) to screen coordinates (pixels). The first two parameters are the X and Y problem coordinates of a point, and the third parameter is a pointer to the **XPoint** structure in which it stores the corresponding screen coordinates.

Compiling and Linking X Window System Applications

To compile and link an X Window System application for the nodes, use the **icc** command with the **-nx** or **-lnx** switch and one or more of the **-l** switches shown in Table 1-1 and Table 1-2. Other compiler switches may be used as well. For information on the switches that the compiler accepts, refer to the *Paragon™ OSF/1 C Compiler User's Guide.*

Some X libraries depend on other libraries. You must specify them in the following order on the command line:

```
-lXaw -lXmu -lXt -lXext -lX11
```

If you use any library in this list, you must also include the libraries to its right. If you use any X libraries other than those on this list, place them to the left of this list.

For example, to compile and link a program that uses the Athena widgets (*Xaw*), use a command line like the following:

```
icc -nx filename -lXaw -lXmu -lXt -lXext -lX11
```

To compile the same program for a service node, use the same command line without the **-nx** switch.

Basic X Window System Libraries

Nine X client libraries are provided: five basic libraries and four advanced libraries. The five basic libraries (*Xlib*, *Xaw*, *Xmu*, *Xt*, and *oldX*) are documented in the X Window System manuals by O'Reilly and Associates. These libraries and the volumes in which they are documented are listed in Table 1-1. The *Xlib* library is the only one whose name on the system is different from the standard library name.

Table 1-1. Basic X Window System Libraries

Library Name	Function and Documentation	Link Switch
Xlib	Core X Window System library (Volumes 1 and 2 of the O'Reilly manuals)	**-lX11**
Xaw	Athena widget set (Volumes 4 and 5 of the O'Reilly manuals)	**-lXaw**
Xmu	MIT miscellaneous utilities (Volume 2 of the O'Reilly manuals)	**-lXmu**
Xt	Toolkit intrinsics layer (Volumes 4 and 5 of the O'Reilly manuals)	**-lXt**
oldX	X10 compatibility library (Volume 2 of the O'Reilly manuals)	**-loldX**

Advanced X Window System Libraries

The other four supplied libraries (*Xau*, *Xdmcp*, *Xext*, and *Xinput*) are typically used for advanced X Window System programming. Documentation for these libraries is supplied in **troff** format and is located in the directory */usr/lib/X11/doc* on the Intel supercomputer. Specific directories for these documents within */usr/lib/X11/doc* are shown in Table 1-2.

Table 1-2. Advanced X Window System Libraries

Library Name	Function and Documentation	Link Switch
Xau	X authorization protocol (troff documentation in */usr/lib/X11/doc/Xau*)	**-lXau**
Xdmcp	X display manager control protocol (troff documentation in */usr/lib/X11/doc/Xdmcp*)	**-lXdmcp**
Xext	Miscellaneous X extensions (troff documentation in */usr/lib/X11/doc/Xext*)	**-lXext**
Xi	X input extension (troff documentation in */usr/lib/X11/doc/Xinput*)	**-lXi**

Problems in Opening the Display

This section describes what to do if you see the following message when you try to run an X program on the Intel supercomputer:

```
Error: Can't Open display
```

This message indicates that the node program has failed to open a connection with the server over the network. Three of the most common causes of this message are:

- You have not told the node program which server to use.
- The Intel supercomputer and the server do not know each other's IP address.
- The Intel supercomputer is not authorized to access the server.

The following subsections describe these problems and their solutions.

Specifying the Server to the Node Program

There is no X Window System server on the Intel supercomputer. Therefore, you must always specify the display when you run an X program. The default value **unix:0** will not work on the Intel supercomputer.

You can specify the display with the **-display** command-line argument or by setting the *DISPLAY* environment variable on the Intel supercomputer to the appropriate value for your server. Use the following command on the Intel supercomputer to check the value of your *DISPLAY* variable (if you use a shell other than **csh**, use the appropriate commands):

```
echo $DISPLAY
```

The correct value is usually the name of the server computer followed by ":0". For example, suppose your X server is a workstation called *mysun*. Before loading a node program that makes X client calls, issue the following command:

```
setenv DISPLAY mysun:0
```

To have the *DISPLAY* variable set automatically every time you log in, put the appropriate **setenv** command in your *.cshrc* or *.login* file on the Intel supercomputer, entering a line like the following:

```
setenv DISPLAY mysun:0
```

This line ensures that all programs started from the Intel supercomputer, including node programs, use *mysun* as the X server. If your X server has more than one display, you may use the following form:

```
setenv DISPLAY mysun:0.0
```

Ensuring that Supercomputer and Server Know Each Other's Address

If you have specified the server to the X program but you still get the `Can't Open display` error message, you must make sure that the Intel supercomputer and the server know each other's network address.

1. Use the **ping** command on the Intel supercomputer to check that it knows the address of the server. You need to use the full pathname for the **ping** command, as in this example:

```
/sbin/ping mysun
PING mysun.myco.com (012.34.567.890): 56 data bytes
64 bytes from 012.34.567.890: icmp_seq=0 ttl=255 time=10 ms
64 bytes from 012.34.567.890: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 012.34.567.890: icmp_seq=2 ttl=255 time=0 ms
<Ctrl-c>

----mysun.myco.com PING Statistics----
3 packets transmitted, 3 packets received, 0% packet loss
round-trip (ms) min/avg/max = 0/3/10 ms
```

The above output indicates that the Intel supercomputer knows the address of *mysun*. The following output indicates that it does not know the address:

```
ping: unknown host mysun
```

If you see this message, ask your system administrator to add the server's name and address to the Intel supercomputers */etc/hosts* file, or your site's NIS database. If the **ping** command hangs, it may indicate that the specified name is in the */etc/hosts* file but the address is wrong; the system administrator can check this.

2. If the Intel supercomputer knows the server's address, check whether the server knows the Intel supercomputer's address. To check this, you must know the Intel supercomputer's *hostname*. If you do not know this name, use the **hostname** command on the Intel supercomputer. For example:

```
hostname
super
```

3. Once you know the hostname, use **ping** or the equivalent command on the server to check that the server knows the Intel supercomputer's address. On a Sun system, **ping** is not in the default execution path for users other than *root*, so you must specify its full pathname. For example:

```
mysun% /usr/etc/ping super
super is alive
mysun%
```

The output above indicates that *mysun* knows the address of *super*. The following output indicates that it does not know the address:

```
ping: unknown host super
```

If you see this message, your system or network administrator needs to add the Intel supercomputer's name and address to the server's */etc/hosts* file or NIS database.

Authorizing the Supercomputer to Access the Server

If the Intel supercomputer knows the address of the server, but the server does not authorize access by the Intel supercomputer, the following messages may appear:

```
Xlib:  connection to "mysun:0.0" refused by server
Xlib:  Client is not authorized to connect to Server
Error: Can't Open display
```

The following steps describe how to fix this problem for **xhost** authorization, the most common authorization system. Consult your system administrator if your system uses a different kind of authorization.

1. Determine the Intel supercomputer's hostname, as explained in the previous section.

2. Once you know the hostname, use the **xhost** command on the server to allow access to that name:

```
mysun% xhost super
super being added to access control list
mysun%
```

3. To list the authorized hosts, issue the **xhost** command with no arguments:

```
mysun% xhost
access control enabled (only the following hosts are allowed)
super
bear
wolf
localhost
mysun%
```

The effect of the **xhost** command lasts only as long as the X server software is running, so you might want to add this **xhost** command to your *.xinitrc* file on the server.

Using the Distributed Graphics Library 2

The Distributed Graphics Library (DGL) is a software library of subroutines developed by Silicon Graphics, Inc. (SGI) for two-dimensional and three-dimensional graphical programming. It can control the displays of workstations, and it provides a standard environment for application software.

A set of DGL client libraries is offered as an option under the Paragon OSF/1 operating system. These libraries run either in service or compute partition of the Intel supercomputer. Applications using DGL may be written in either Fortran or C.

This chapter describes:

- Special programming techniques for node DGL programs
- How to compile and link DGL applications
- Where node DGL programs look for resources
- What to do if your node DGL program cannot open the display server.

This chapter only includes information specific to writing DGL applications for Paragon OSF/1. It does not describe how to write DGL applications programs. For information on writing DGL programs, refer to the SGI *Graphics Library Programming Guide*.

To use your workstation as a server with access to the Paragon OSF/1 DGL client libraries, the DGL daemon, */usr/etc/dgld*, must be activated on your workstation. For information on how to do this, refer to "Using the Network-Transparent Feature of GL" on page 2-13.

An entry for your DGL server must be included in the Intel supercomputer's */etc/hosts* file or NIS database. If no such entry exists, your system administrator must add an entry for your server. Refer to "Ensuring that the System and Server Know Each Other's Address" on page 2-11 for more information.

Most of the programming techniques and considerations described in this chapter apply to programs to be run in the compute partition. If you create a DGL program to run only in the service partition, no special programming techniques are necessary. You need only link service partition applications properly, as described in "Compiling and Linking DGL Applications" on page 2-10.

A Sample DGL Program

NOTE

The sample program described in this chapter is not yet available on the system. The description of how to use it is included in this release for informational purposes. An example will be available on line in a future software release.

What the *graph* Program Does

The *graph* program performs a simple calculation for the value of **sin**(x) for $0 \le x < 2\pi$ and graphs the result in a window as it is calculated. You can resize the window, using your window manager. The graphed image remains proportional to the size of your window, growing as the window enlarges, shrinking as the window gets smaller. A **restart** button, when selected, clears the screen and restarts the calculation from the beginning. A **quit** button allows you to terminate the program.

The problem decomposition used in this example is a modified domain decomposition. Node 0 maintains the display, and the other nodes calculate the points of the curve. The x values from 0 to 2π are divided evenly among the calculating nodes. For example, if the program is run on four nodes, node 1 is responsible for $0 \le x \le 2\pi/3$, node 2 is responsible for $2\pi/3 < x \le 4\pi/3$, and node 3 is responsible for $4\pi/3 < x < 2\pi$.

Figure 2-1 shows what the *graph* program looks like when running. (Your window manager might also place a header and/or border on the window, which are not shown here.)

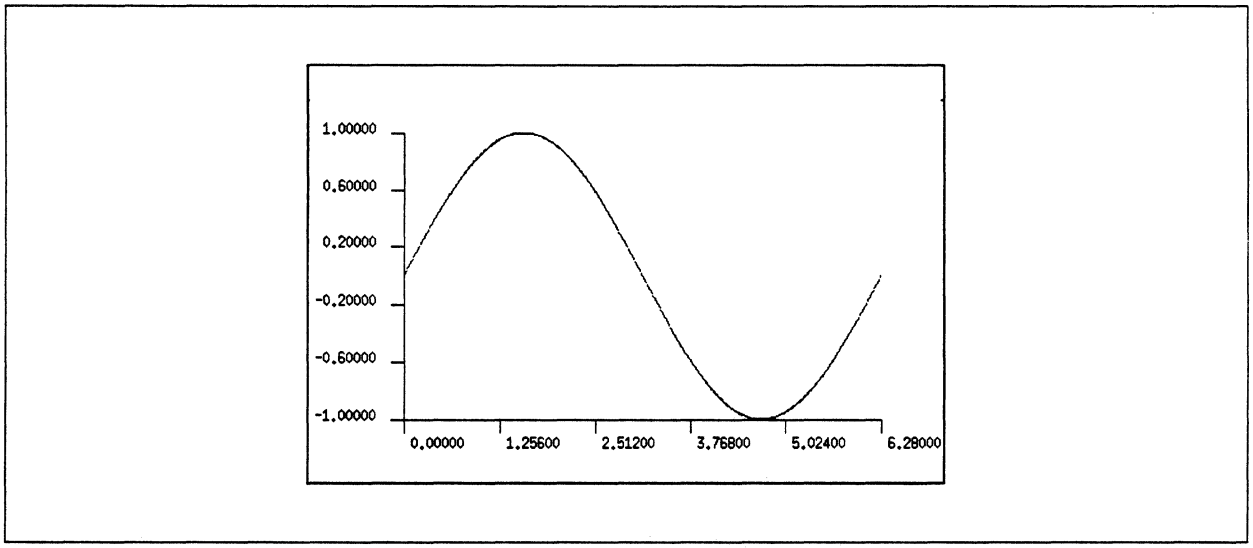


Figure 2-1. *graph* Program Display

Compiling, Linking, and Executing the *graph* Program

This section describes how to compile and link the C or Fortran language versions of the *graph* program. Your Intel supercomputer system software must include the DGL option. To compile these programs on your workstation, you must have the **icc** and/or **if77** cross-compiler.

NOTE

Because the C and Fortran versions of this program are not yet available on line, these instructions list the variable *path* in place of an actual path. When available, these programs and their makefiles will reside in separate directories.

1. Create a directory for the *graph* program in your current directory with the following command:

   ```
   mkdir graph
   ```

2. Copy the source code and *makefile* for the *graph* program to your *graph* directory.

   On the Intel supercomputer, use the following command:

   ```
   cp path/graph/* graph
   ```

On a workstation, use either **rcp** or **ftp** to transfer the file from the supercomputer to your workstation. For example, you could use **rcp**, as in the following command, replacing *super* with the name of your Intel supercomputer:

```
rcp "super:path/graph/*" graph
```

3. Use the following command to change to the *graph* program directory:

```
cd graph
```

4. Use the following command to compile and link the program:

```
make
```

5. If you compiled the program on a workstation, copy the executable to the Intel supercomputer and then log into the Intel supercomputer, using the appropriate command for your site. For example, if the appropriate commands are **rcp** and **rlogin**, use commands like the following:

```
rcp graph super:
rlogin super
```

6. When the program has been compiled and linked, set the *DGLSERVER* environment variable to the appropriate value for your server. For example, if your shell is **csh** and your server is a workstation called *mysgi*, use the following command:

```
setenv DGLSERVER mysgi
```

7. Run the program on at least two nodes; four or more are recommended. For example, to run the program on eight nodes of your default partition, you would use the following command:

```
graph -sz 8
```

The *graph* window appears and the graph is drawn. If the message "Error: Can't Open display" appears instead, refer to "Problems Opening the Display" on page 2-10.

For information on controlling the execution of parallel applications, refer to the *Paragon™ OSF/1 User's Guide*.

8. Click the button labeled **restart** if you want to draw the graph again. If you want to quit, click the **quit** button. These buttons are not shown in Figure 2-1, but appear on your display.

Flow of Control in the *graph* Program

Figure 2-2 shows a flow chart for the *graph* program.

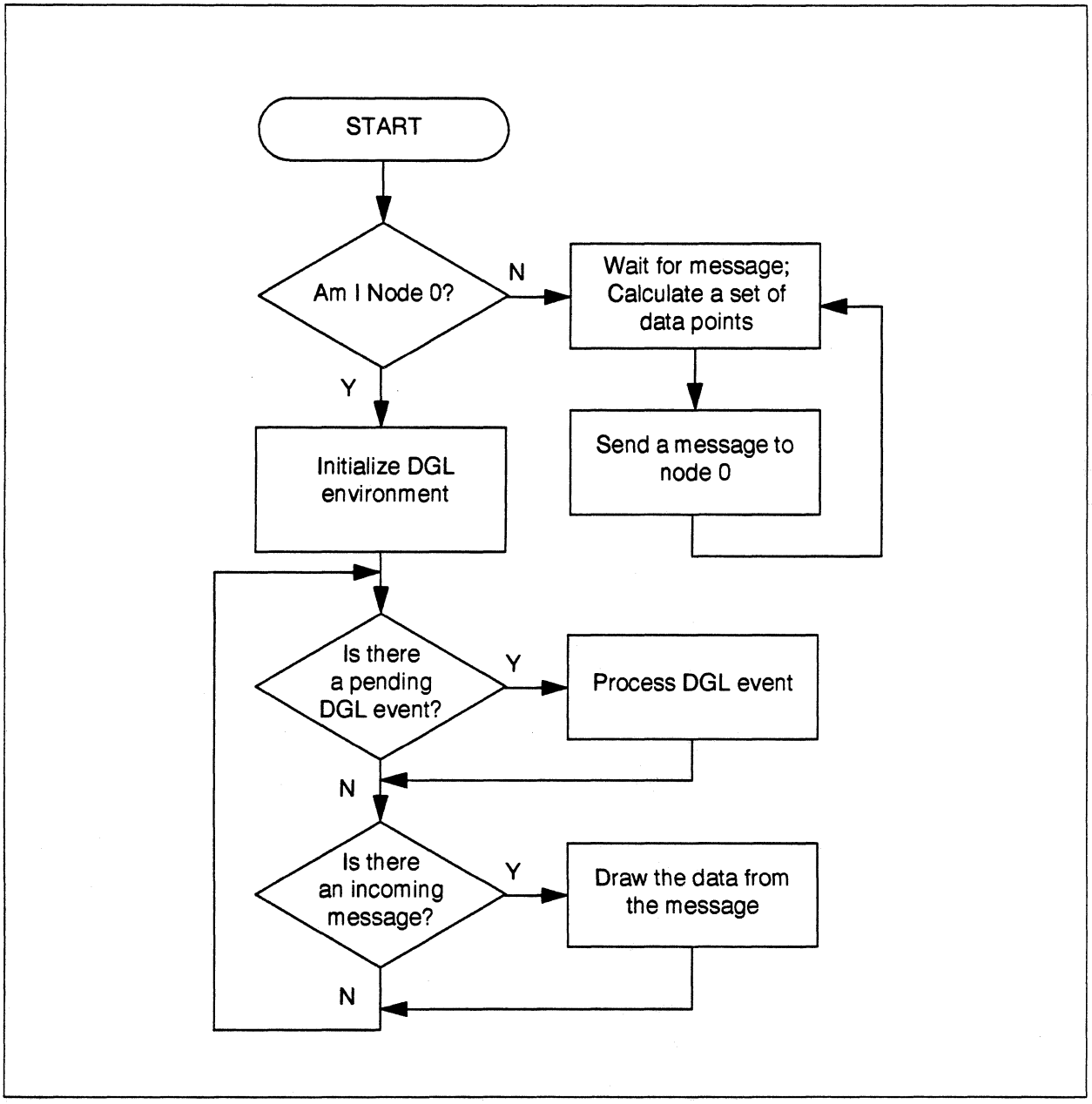


Figure 2-2. Flow Chart of the *graph* Program

Node 0 sets up the display window and all of the pop-up menus, and then loops while awaiting a DGL event or a message. When this node receives an event, the event is processed. When this node receives a message, it graphs the data found in the message.

Nodes other than node 0 just calculate the points of the graph and send them to node 0 as messages.

Programming Techniques

This section discusses the special programming techniques for writing DGL applications that run on multiple nodes. These techniques are demonstrated in the *graph* program.

Connecting Nodes to the Server

In the *graph* program, only node 0 makes DGL calls. The other nodes only calculate; they pass the results of their calculations to node 0 as messages. Node 0 graphs the contents of each message it receives.

If you have a single node handle the display, you may to have this node calculate as well, depending on the relative importance of speedy calculation and good graphics performance for your application. In the *graph* program, node 0 does not calculate; it only handles the display. Using node 0 strictly to handle the display and not to calculate is simpler to program, but produces no results unless run on two or more compute nodes.

Combining DGL Event-Driven Programming with Message Passing

DGL programs are built around a "main event loop", which repeatedly receives and then manipulates a DGL event. The problem with using a such loop in a message-passing node program is that the program blocks while waiting for the next event (**qread**() does not return until there is an event). Therefore, when no DGL events are coming in, the node is blocked; it cannot calculate or handle messages from other nodes.

By using a non-blocking call in your main event loop, you can let the loop proceed rather than block if there is no DGL event pending. In a DGL program, you would call **qtest**() instead of **qread**().

Similarly, you should use non-blocking message calls, such as **irecv**(), so neither the message-handling part of the program nor the DGL event-handling part blocks the other.

In the C version of the *graph* program, the main event loop looks like the following, with the call of **qtest()** instead of the usual call to **qread()**:

```
/* prepare to receive first message */
    msgid = irecv(DATA, &points, sizeof(points));

    /* start the nodes */
    csend(START, NULL, 0, -1, 0);

    /* infinite loop for DGL events and messages */
    while(1) {
        if(qtest() != 0)
            HandleDGLEvent();
        if(msgdone(msgid)) {
            int i;

            /* insert the new line segments in the object */
            editobj(Wave);
            bgnline();
            for(i=0; i<points.npoints; i++)
                v2f(points.points[i]);
            endline();
            closeobj();

            /* draw the new data */
            callobj( Wave );

            msgid = irecv(DATA, &points, sizeof(points));
        }
    }
```

The main event loop of the Fortran version of *graph* looks like this:

```
C
C  prepare to receive first message
C
      msgid = irecv(DATA, msgbuf, 4*MSGSZ)
C
C  start the nodes
C
      call csend(BEGIN, points, 0, -1, 0)
C
C  infinite loop for DGL events and messages
C
1000  if( qtest() .NE. 0) then
            call dglevt(wave, mymenu)
      else if( msgdone(msgid) .NE. 0) then
            call msgevt(wave, npoints, vector)
```

```
        msgid = irecv(DATA, msgbuf, 4*MSGSZ)
    endif
    goto 1000
```

Responding to Window Destruction

If the user destroys your application's window with the window manager's "Kill Window" function, the default window destruction procedures terminate the DGL node. However, this does *not* terminate the calculating nodes.

The effect of this problem depends on your application. If this is a problem for your application, you should add cases to your DGL event loop that catch **WINQUIT** and **WINSHUT** events and, if found, call **kill(0, SIGKILL)**) to terminate the compute nodes.

Batching Data Points into Larger Messages for Improved Performance

Graphics performance usually suffers when you draw only one point at a time. The performance of a message-passing program also suffers when you spend a lot of time passing messages. You can improve both kinds of performance by batching your data points into larger messages.

The C code that collects the data into messages in the *graph* program looks like this:

```
struct {
    int npoints;
    float points[NPOINTS][2];
} points;       /* data points */
          .
          .
          .
    Coord x, unit, start, end;
    int i;
          .
          .
          .
          for(x = start; x < end; x += STEP) {
              points.points[i][0] = x;
              points.points[i][1] = sin(x);
              i++;
              if(i >= NPOINTS) {
                  points.npoints = i;
                  csend(DATA, &points, sizeof(points),
                        0, 0);
                  i = 0;
              }
          }
```

```
        if(i != 0) {
            points.npoints = i;
            csend(DATA, &points, sizeof(points), 0, 0);
        }
```

The Fortran code that collects the data into messages in the *graph* program looks like this:

```
      integer*4 npoints
      real*4 vector(2,100)
      real*4 msgbuf(201)
      equivalence(npoints,msgbuf(1))
      equivalence(vector(1,1), msgbuf(2))
         .
         .
         .
      real*4 x, start, end
      integer*4 i
         .
         .
         .
         i = 1
         DO 2000 x = start, end, STEP
            vector(1,i) = x
            vector(2,i) = sin(x)
            i = i + 1
            if(i .GT. NPTS ) then
               npoints = i - 1
               call csend(DATA, msgbuf, 4*MSGSZ, 0, 0)
               i = 1
            endif
2000     continue

         if(i .NE. 1) then
            npoints = i - 1
            size = 4 * ( (2 * npoints) + 1)
            csend(DATA, msgbuf, size, 0, 0)
         endif
```

Compiling and Linking DGL Applications

To compile and link a DGL application for compute nodes, use the **icc** command or the **if77** command with the **-nx** switch, the **-ldgl** (for both C and Fortran) and **-lfgl** (for Fortran only) switches. The **-nx** switch compiles the code to be executed on compute nodes. The node TCP/IP library required for all DGL programs is included automatically, with no extra switches required in the compile command line.

- To compile and link a C program for the compute partition, use a command line like the following:

```
icc -nx filename -ldgl
```

- To compile and link a Fortran program for the compute partition, use a command line like the following:

```
if77 -nx filename -lfgl -ldgl
```

- To link for the service partition, use the same command lines as above, but without the **-nx** switch.

Problems Opening the Display

This section describes what to do if you see an error message when you try to run a DGL program on an Intel supercomputer. When a node program has failed to open a connection with the server over the network, one of the following three problems is likely to be the cause. Error messages that usually indicate one of these problems are also shown.

- You have not told the node program which server to use.

```
libdgl error (pipe_init): DGLLOCAL not supported
libdgl error (default init): default dglopen returned -238436736
```

- The Intel supercomputer and the server do not know each other's Internet address.

```
libdgl error (*gethostbyname): can't get addr for name
libdgl error (write): value
```

- The Intel supercomputer is not authorized to access the server.

```
libdgl error (login): dgl server access denied -
Cannot open link to DGL server mysgi
```

The following subsections describe these problems and their solutions.

Specifying the Server to the Node Program

There is no DGL server on the Intel supercomputer, and no default server value for the Intel supercomputer. As a result, you must always specify the display when you run a DGL program

You specify the display by setting the *DGLSERVER* environment variable on the service node to the appropriate value for your server. (Environment variables are automatically copied from the service node shell to the node program.) Use the following command on the service node to check the value of your *DGLSERVER* variable:

```
echo $DGLSERVER
```

The correct value is usually the name of the server computer. For example, suppose your DGL server is a workstation called *mysgi*. Before loading a node program that makes DGL client calls, issue the following command:

```
setenv DGLSERVER mysgi
```

To set the *REMOTEHOST* variable automatically every time you log in, put the appropriate **setenv** command in your *.cshrc* or *.login* file on the service partition. For example:

```
setenv REMOTEHOST mysgi
```

For more information, refer to "Establishing a Connection" on page 2-15.

Ensuring that the System and Server Know Each Other's Address

If you have specified the server to the DGL program, but the Intel supercomputer and the server do not know one another's addresses, you may still get one of the following error messages:

```
libdgl error (*gethostbyname): can't get addr for
libdgl error (write):
```

To ensure that the Intel supercomputer and the server know each other's network address, perform these steps:

1. Use the **ping** command on the Intel supercomputer to check that it knows the address of the server. For example:

```
/sbin/ping mysgi
PING mysgi.myco.com (012.34.567.890): 56 data bytes
64 bytes from 012.34.567.890: icmp_seq=0 ttl=255 time=10 ms
64 bytes from 012.34.567.890: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 012.34.567.890: icmp_seq=2 ttl=255 time=0 ms
<Ctrl-c>

----mysgi.myco.com PING Statistics----
3 packets transmitted, 3 packets received, 0% packet loss
round-trip (ms) min/avg/max = 0/3/10 ms
```

   The output above indicates that the Intel supercomputer knows the address of *mysgi*. The following output indicates that it does not know the address:

```
ping: unknown host mysgi
```

   If you see this message, ask your system administrator to add the server's name and address to the Intel supercomputers */etc/hosts* file, or your site's NIS database.

   If the **ping** command hangs, it may indicate that the specified name is in the */etc/hosts* file but the address is wrong; the system administrator can check this.

2. If the Intel supercomputer knows the server's address, check to see that the server knows the supercomputer's address. Use the **ping** command on the SGI workstation to verify this, as in the following example, which assumes the name of the Intel supercomputer to be **super**:

```
mysgi% /usr/etc/ping super
super is alive
mysgi%
```

   The preceding output above indicates that *mysgi* knows the address of *super*. The following output indicates that it does not know the address:

```
ping: unknown host super
```

   If you see this message, have your system or network administrator add the Intel supercomputer system's name and address to the server's */etc/hosts* file or NIS database.

Authorizing the Supercomputer to Access the Server

If you see the following messages when you try to run a DGL program, it means that the Intel supercomputer system knows the server's address, but the server does not authorize the Intel supercomputer system to access it:

```
libdgl error (login): dgl server access denied -
Cannot open link to DGL server mysgi
```

Use the appropriate command on the server to allow access. The procedure for doing this is described in the next section.

Using the Network-Transparent Feature of GL

NOTE

Most of the information in this section appears as Chapter 19 in the *Graphics Library Programming Guide* from Silicon Graphics, Inc. (Document Number 007-1210-040)

Writing a network-transparent GL program is no different than writing a standalone GL program, except for optimizing performance. Graphics calls are buffered from the client to the server, so you must flush the buffer periodically. The subroutine **gflush**() flushes the client buffer so the server can receive GL calls.

The gflush() Subroutine

The DGL client buffers calls to GL subroutines for efficient block transfer to the graphics server. The subroutine **gflush**() explicitly flushes the communication buffers and delivers all the untransmitted graphics data that is in the buffer to the graphics server.

GL subroutines that return data implicitly flush the communication buffers. In most programs, the implicit flushing that is performed by subroutines that return data is usually sufficient.

NOTE

All programs that are run over the network must call **gflush()** if the last command is a drawing command. No drawing is guaranteed to happen until **gflush()** is called.

The following example outlines a typical use of **gflush**():

> A program calls some Graphics Library subroutines that are buffered and not flushed. The program then either computes or blocks for a while, waiting for non-graphic I/O. The **gflush**() subroutine must be called if the results of the buffered GL subroutines are to be seen on the host display before and during the pause.

Another reason for using **gflush**() is to reduce graphics "jerkiness." If the client is computing data and then sending that data to the graphics server without implicit or explicit flushes, the data will arrive at the graphics server in large batches. The server may process this data very quickly and then wait for the next large batch of data. The rapid processing of GL subroutines followed by a pause results in an undesirable "jerky" appearance. In these cases it is probably best to call **gflush**() periodically. For example, a logical place to call **gflush**() is after every **swapbuffers**() call.

NOTE

Performing too many flushes can adversely affect performance.

The finish() Subroutine

The **finish**() subroutine is useful when there are large network and pipeline delays. The **finish**() subroutine blocks the client process until all previous subroutines execute. First, the communication buffers on the client machine are flushed. On the graphics server, all unsent subroutines are forced down the Geometry Pipeline to the bitplanes, then a final token is sent and the client process blocks until the token goes through the pipeline and an acknowledgment is sent to the graphics server and forwarded to the client process.

The following example illustrates a typical use of **finish**:

> A client calls GL subroutines to display an image. The subroutines all fit into the server's network buffers and the image takes 30 seconds to render. The client wants to wait until the image is completely displayed on the server's monitor before a message can be displayed on the client's terminal. The **gflush**() subroutine flushes the buffers, but does not wait for the server to process the buffers. The **finish**() subroutine flushes the buffers and waits not only for the server to process all the graphics subroutines, but for the Geometry Pipeline to finish as well.

Establishing a Connection

The DGL server is initialized by the routine **dglopen**(). If **dglopen**() is not called by the program, the DGL library attempts to open a default connection by calling **dglopen**() with a default server name and connection type. If either of the following environment variables are defined, the server name is the value of the defined variable highest in the following list:

- *DGLSERVER*
- *REMOTEHOST*

If the value of *REMOTEHOST* is used for the server name, then the environment variable *REMOTEHOSTUSER* is checked. If *REMOTEHOSTUSER* is defined, the server name is set to `REMOTEHOSTUSER@REMOTEHOST`. If neither of the environment variables above are defined, then the server name is set to the client's hostname.

The value for the connection type comes from the following ordered list:

1. *DGLTYPE* (if defined)
2. *DGLSOCKET* (if an environment variable is used for the server name)
3. *DGLLOCAL*

The environment variable *DGLTYPE* can be set to either the symbolic or numeric value of the connection type, for example, *DGLSOCKET* or 2.

Limitations and Incompatibilities

The network-transparent GL had a few limitations and incompatibilities with the previous releases of the GL, which was used strictly for local imaging. These limitations may prevent a GL application from executing properly when remote connections are used.

The callfunc() Subroutine

The **callfunc**() subroutine does not function in a GL program that is run remotely. Any references to **callfunc**() will result in a runtime error when loading the program.

Pop-up Menu Functions

A maximum of 16 unique callback functions are supported. Freeing pop-up menus does not free up callback functions. If you use too many callback functions, you get the following client error:

```
dgl error (pup): too many callbacks
```

Interrupts and Jumps

You cannot interrupt the execution of a remotely called GL subroutine before calling another subroutine. This typically happens when you set an alarm or timer interrupt to go off and then block the program with a **qread()** call. If the signal handler does not return to the **qread()**, unpredictable results are likely; for example, it could do a **longjump()** to some non-local location.

DGL Configuration

The DGL protocol software consists of two parts: a client library and a graphics server daemon. The graphics server daemon is */usr/etc/dgld*. The DGL protocol gets an Internet port number from */etc/services*, which is set up during installation of DGL to have an entry for *sgi-dgl* (see the **services()** online manual page).

The inetd Daemon

The graphics server daemon for TCP socket connections is automatically started by **inetd**. This command reads its configuration file to determine which server programs correspond to which sockets. The standard configuration file, */usr/etc/inetd.conf*, has an entry for *sgi-dgl*. When a request for a connection is made the following sequence occurs:

1. The service *sgi-dgl* is looked up in */etc/services* to get a port number. If the service is not found, an error occurs.

2. The server's name is looked up in */etc/hosts* to get an Internet address. If the host is not found, an error occurs.

3. An Internet stream socket is created and some of its options are set.

4. A connection to the server machine is attempted with a small timeout allowance. If the connection is refused, the timeout is doubled and the connection retried. If after several tries the connection is still refused, an error occurs.

5. A successful connection is made and the server's Internet daemon invokes a copy of the DGL graphics server. The graphics server process inherits the socket for communicating with the DGL client program.

6. The graphics server uses the **ruserok**() call to verify the login. The user ID on the server must be the equivalent (in the sense of **rlogin**) to the user ID running the DGL client program or permission is denied.

7. The server process's group and user IDs are changed according to the entry in */etc/passwd*.

The dgld Daemon

The **dgld** daemon is the server for remote graphics clients. The server provides both a subprocess facility and a networked graphics facility. The **dgld** daemon is started by **inetd** when a remote request is received.

TCP socket connections are serviced by the Internet server daemon **inetd**. **inetd** listens for connections on the port indicated in the *sgi-dgl* service specification. When a connection is found, **inetd** starts **dgld** as specified by the file */usr/etc/inetd.conf* and gives it the socket.

Error Messages

Error messages are output to a message file. The message file defaults to *stderr*. Error messages have the following format:

```
pgm-name error (routine-name): error-text
```

where:

pgm-name Either `dgl` for client errors or `dgld` for server errors.

routine-name The name of the system service or internal routine that failed or detected the error.

error-text An explanation of the error.

Connection Errors

Table 2-1 lists the internally generated error values that are reported when a connection fails.

Table 2-1. Connection Error Values

Error Value	Explanation
ENODEV	Type is not a valid connection type.
EACCESS	Login incorrect or permission denied.
EMFILE	Too many graphics connections are currently open.
ENOPROTOOPT	DGL service not found in */etc/services*.
ENPROTONOSUPPORT	DGL version mismatch.
ERANGE	Invalid or unrecognizable number representation.
ESRCH	Window manager is not running on the graphics server.

Client Errors

Client error messages are printed to *stderr*. For example, if NIS is not enabled and */etc/hosts* does not include an entry for the server host *foobar*, the following error message is printed when a connection is requested:

```
dgl error (gethostbyname): can't get name for hostname
```

If the client detects a condition that is fatal, it makes an **exit()** call, with an *errno* value as its parameter that best indicates the condition. If a system call or service returns an error number (*errno* or *h_errno*), this number is used as the exit number.

Table 2-2 lists all exit values that are internally generated (not the result of a failed system call or service).

Table 2-2. GL Client Exit Values

Exit Value	Explanation
ENOMEM	Out of memory.
EIO	Read or write error.

The EIO value is sometimes accompanied by the following message:

```
dgl error (comm): read returned 0
```

This means that the communication with the server has been interrupted or was not successfully established. The configuration of the server machine should be checked (see "DGL Configuration" on page 2-16).

Server Errors

Server error messages are printed to *stderr* by default. For example, if */etc/hosts* does not include an entry for the client host, the following error messages appear:

```
dgl error (gethostbyaddr): can't get name for 59000002
dgl error (comm_init): fatal error 1
```

The standard *inetd.conf* file runs the graphics server with the **I** and **M** options. The **I** option informs the graphics server that it was invoked from **inetd** and enables output of all error messages to the system log file maintained by **syslogd**. The **M** option disables all message output to *stderr*.

If the DGL server is not working properly, check the system log file (*SYSLOG*) for error messages (see your system administrator for its location). Each entry in the *SYSLOG* file includes the date and time of the entry, identifies the program as **dgld**, and includes the process identification number (PID) for the server process. The rest of the error message is the text of the error message.

Exit Status

When the **dgld** graphics server exits, the exit status indicates the reason for the exit. A normal exit has an exit status of zero. A normal exit occurs when either the client calls **dglclose()** or when zero bytes are read from the graphics connection. The latter case can occur when the client program exits without calling **dglclose()** or terminates abnormally.

A non-zero exit status implies an abnormal exit. If the graphics server program detects a condition that is fatal, it exits with an *errno* value that best indicates the condition. If a system call or service returned an error number (*errno* or *h_errno*), this number is used as the exit number.

Table 2-3 lists all exit values that are internally generated (not the result of a failed system call or service).

Table 2-3. GL Server Exit Values

Exit Value	Explanation
0	Normal exit
ENODEV	Invalid communication connection type
ENOMEM	Out of memory
EINVAL	Invalid command line argument
ETIMEDOUT	Connection timed out
EACCESS	Login incorrect or permission denied
EIO	Read or write error
ENOENT	Invalid GL routine number
ENOPROTOOPT	DGL/TCP service not found in */etc/services*
ERANGE	Invalid or unrecognizable number representation

The Parallel Make Utility 3

The parallel **make** utility **pmake** for Paragon OSF/1 brings the advantages of parallel processing to a traditionally time-consuming part of program development—building and updating programs that consist of multiple source files. The **pmake** utility is based on GNU **make**, the **make** utility written by Richard Stallman and Roland McGrath. The **pmake** utility, in addition to the features of GNU **make**, gives you control over parallel execution in the compute partition of the Intel supercomputer, and many other features designed to improve compatibility with other **make** utilities.

NOTE

pmake is an extension of GNU make and is distributed under the terms of the GNU General Public License. As such, Intel will provide a complete, machine-readable copy of the **pmake** source code upon request. For more information, contact Intel's Customer Service Response Center, as described in the section "Comments and Assistance" in the Preface of this book.

Most computer applications consist of numerous source modules, each of which may refer to one or more include files. Whenever any of these files is changed during the development process, the following must occur:

- Each changed file must be recompiled.
- All files that depend upon the changed file must be updated.
- All of the files must be relinked to update the application.

The purpose of a **make** utility is to make this process as automatic and as efficient as possible. Generally, the utility is used to recompile large programs, but it can be used for any task where some files must be updated automatically whenever the files that they depend upon change.

This chapter provides an overview of the makefile description file and describes differences between **pmake** and GNU **make**. For detailed information on GNU make, refer to *GNU Make, A Program for Redirecting Compilation.*

The makefile Description File

To use **pmake**, you need a *makefile*, also called a *description file*. A *makefile* can contain four types of statements:

- Rules
- Variable definitions
- Directives
- Comments

The *rules* define when and how to update files (called *targets* of the rules). Rules usually have a single target. A rule lists any files that the targets depend upon, called *dependencies* of the target, and *commands* to be used to create or update the targets. When you have defined your *makefile*, building or rebuilding your application requires only that you enter a single command line, using the **pmake** command.

Rather than simply executing all of the rules in a given *makefile* each time, **pmake** can also determine which source files have been changed since the last update. This allows **pmake** to update only those files and any files that depend upon the changed files, saving computer time when only a few of the files of an application must be updated. This also saves the time of the programmer, who does not need to keep track of all of the details to ensure that a build is done correctly each time.

You can define variables in a *makefile*. A *variable definition* in a *makefile* assigns a text string to a variable, allowing you to use that variable name in place of the complete text string. For example, you could define a variable to be a list of all of the object files.

Directives are special commands. Available directives include directives that read another *makefile* and conditional directives that determine whether parts of the *makefile* are read based on the value of variables.

Comments are allowed in makefiles. A "#" in a line starts a comment; that character and subsequent characters in the line are ignored. You can continue a comment across multiple lines if the last character in a comment line is a backslash (\). A comment cannot be placed within a **define** directive, and within some commands where the shell determines what a comment is. Comments can be in all other *makefile* lines.

For complete information on *makefile* statements and how to construct and use a *makefile*, refer to the *GNU Make* manual.

The **pmake** utility, while based mainly on GNU **make**, offers some differences. This chapter describes these differences, which include an additional parallel execution control, some extensions to macro definition, support of a configuration file, and others. For more information on **pmake**, see the **pmake** manual page, either using the online **man** command, or in the *Paragon™ OSF/1 Commands Reference Manual.*

Parallel Controls

The **pmake** utility is designed to update multiple target files in parallel. Parallel execution can occur in either the service partition or the compute partition. You request parallel execution by using either the **-j** or **-P** switch. You can also use both switches together. The **-P** switch specifies the partition in which **pmake** runs jobs. The default is the service partition.

Using a Compute Partition

When you use the **-P** switch to specify one of the compute partitions on the Intel supercomputer, the **pmake** process calls **nx_initve()** to become a gang-scheduled parallel application. Then the **pmake** process, running in the service partition, acts as the controlling process, sending commands out in parallel to nodes in the compute partition as the nodes become available.

The **-j** switch allows you to specify the maximum number of jobs that can run in parallel. If you do not use the **-j** switch, the maximum number of jobs defaults to the number of nodes in the partition, or one node if **pmake** is running in the service partition. If you use the **-j** switch followed by the optional *jobs* argument, **pmake** runs up to the number of jobs specified in parallel. The number of jobs **pmake** can run in parallel is not limited to the number of nodes in the partition, because multiple jobs can run on a node. If you use the **-j** switch without the *jobs* argument, the maximum number of jobs **pmake** can run in parallel is unlimited.

The following examples illustrate the use of the **-j** and **-P** switches with the **pmake** command. The first example runs N jobs in parallel in the compute partition, where N is the number of nodes in the partition.

pmake -P.compute

The next example runs up to ten jobs in parallel in the compute partition. If there are more than ten nodes in the compute partition, only the first ten are used. If there are less than ten nodes, some nodes will run multiple commands at once.

pmake -P.compute -j10

The next example runs as many jobs as possible in the compute partition. If there are twenty commands that can be run in parallel and only five nodes in the compute partition, each node will run four commands.pmake -P.compute -j

pmake -P.compute -j

The **pmake** utility relies on the dependencies defined in the *makefile* to determine the files that can be updated in parallel. These dependency definitions prevent two files, one of which is dependent upon another, from being updated simultaneously. All commands that update a single file are assumed to be sequential, and are run in the order in which they appear in the *makefile*.

For example, if *file2* is dependent upon *file1*, all commands that update *file1* are run sequentially before commands updating *file2* are executed. If there is no dependency between *file2* and *file1*, commands updating *file2* may be run in parallel with commands updating *file1*. It is, therefore, quite important to ensure that your *makefile* clearly defines all dependencies.

Another important consideration concerns the use of the **-P** switch with recursive makes. Specifying the **-P** switch causes **pmake** to become a gang-scheduled parallel application. Therefore, any use of the **-P** switch in subsequent recursive invocations of **pmake** is ignored, because it is already gang-scheduled. Therefore, you need to use the **-P** switch at a level where it can do the most good.

For example, if you make the files in two directories, one with many files and another with a few files, you would do better to invoke parallelism in updating the large directory, rather than at the upper level, where the parallelism would be wasted. Suppose you entered the following **pmake** invocation:

pmake -j2 -P.compute

Used on the following *makefile*, the previous command would update the two targets *big* and *little* simultaneously.

```
all: big little

big:
      cd bigdir; $(MAKE)

little:
      cd littledir; $(MAKE)
```

The target *little* would be updated quickly, while *big*, which involves many compiles, might take several hours to build, and the benefits of parallelism would be lost. In this case, it would be better to invoke the top-level **pmake** without the **-j** and **-P** switches and to use the switches at the second level:

```
all: big little

big:
      cd bigdir; $(MAKE) -j8 -P.compute
little:
      cd littledir; $(MAKE) -j2 -P.compute
```

Using the Service Partition

If you do not use the **-P** switch, **pmake** runs and executes all *makefile* commands in the service partition. In the service partition, **pmake** uses the **fork**() call to start commands simultaneously, and relies on process migration and load balancing to ensure parallel execution. Using the **-j** switch allows you to specify the number of jobs that **pmake** can run in parallel. If you do not use the **-j** switch, **pmake** runs as a single process on one node of the service partition.

Another parallel switch, **-l**, allows you to restrict **pmake** from executing multiple commands in parallel when the system load average gets too high. The *load* argument to the **-l** switch allows you to specify a load average beyond which **pmake** will limit jobs. In this way, you can ensure that the system is not excessively slowed down. The pmake utility always allows one command to execute, even if the load average is over the specified limit. However, if the load average is over the limit and one or more commands are already executing, **pmake** will not start any more commands. Specifying the **-l** switch with no load value removes all previous load limits. The **-l** switch is most useful when running in the service partition, where there is more likely to be contention for system time.

Macro Extensions

The **pmake** utility provides the special macro **$$@** and three macro references (pattern replacement references using regular expressions, C-shell-style modifiers, and conditional expressions) in addition to the macro capabilities of GNU **make**.

- The macro **$$@** provides compatibility with System V **make**,. This macro is used on the dependency line of a rule and is interpreted as the current target (the one currently being processed).

- You can use a pattern replacement reference that causes a replacement string to be substituted for a regular expression within the macro expansion. This has the form:

  $(*MACRO*/*reg-expression*/*replacement*)

  where *reg-expression* is a regular expression, (as described in the online manual page **regexp**()), and *replacement* is the replacement string. The syntax also allows you to use semicolons in place of slashes.

- You can use modifiers similar to the C-shell file name modifiers in variable expressions with the form:

  $(*MACRO*:*X*)

  In this reference, *X* may be **t** (tail), **h** (head), **r** (root) or **e** (extension).

- You can define conditional variables using C-style colon expressions as follows:

  $(*MACRO*?*value1*:*value2*)

  An expression of this form evaluates to *value1* if *MACRO* is defined and *value2* if it is not.

Configuration File Support

The **pmake** utility supports the use of a *configuration file*. This file, if it exists, must be named *Makeconf*; **pmake** searches for it backwards from the current directory to the root directory. Only the first *Makeconf* file found in the path is evaluated, so an empty *Makeconf* file in a searched directory terminates the search. Although **pmake** searches for a *Makeconf* file by default, and does not return an error if no *Makeconf* is found, you can explicitly disable all *Makeconf* processing by using the **-N** switch.

On finding a *Makeconf* file, **pmake** evaluates it as though its contents were at the top of the *makefile*. Although you can define any global variables or other global information in a *Makeconf* file, the file is most useful when your software project is organized into completely separate source and object directory trees.

To support separate source and object directory trees, the *Makeconf* file can define the variable *OBJECTDIR* to be the root of the object directory tree. This can be defined either as an absolute path, or a path relative to the location of *Makeconf*. There is special processing for the *OBJECTDIR* definition. Before running any commands, **pmake** goes to the object directory and modifies its search path to include the path back to the corresponding source directory. For example, suppose a directory named *$SRC* is your root source directory, and that this directory has a *Makeconf* file containing the following absolute path *OBJECTDIR* definition:

```
OBJECTDIR=/topdir/objdir
```

In this case, invoking **pmake** from within the directory *$SRC/subdir* causes **pmake** to create the directory */topdir/objdir/subdir* (if it does not exist), and to use that as the object directory. This ensures that the object directory structure is the same as the source directory structure.

You can also specify the *OBJECTDIR* definition as a pathname relative to the directory containing the *Makeconf* directory, and the directory structure will be created correctly.

Then, after determining and setting the path for the object directory, **pmake** executes *makefile* commands, reading from the source directory, and creating or modifying files in the object directory.

By default, **pmake** looks for source files in the directory containing the *makefile*. You may, however, specify a different source directory by defining the SOURCEDIR variable in the *Makeconf* file. This definition has the following form:

SOURCEDIR=*path1*[**:***path2*]...[**:***pathn*]

As with the *OBJECTDIR* definition, you can define the *SOURCEDIR* path(s) either as absolute or relative paths. As the syntax shows, you can also specify additional source directory trees to be searched. You can assign these alternate source root paths as a colon-separated list to the *SOURCEDIR* variable in the *Makeconf* file. This assignment permits you to work with source files stored in various trees. These paths can also be specified either as absolute paths or as paths relative to the location of the *Makeconf* file.

Other Differences Between pmake and GNU make

There are other minor differences between **pmake** and GNU **make**. These are mainly enhancements to improve compatibility with other **make** programs.

The **pmake** utility supports the following command line switches not supported by GNU **make**:

- The **-c** switch, which causes **pmake** not to try to find and check out a corresponding SCCS or RCS file when a file does not exist.

Note

Paragon OSF/1 currently supports only SCCS, but if you have ported RCS to your Intel supercomputer, **pmake** supports the RCS structure.

- The **-m** switch, which causes **pmake** to search machine-specific subdirectories automatically.
- The **-N** switch, which disables all *Makeconf* processing, as described in "Configuration File Support" on page 3-6.
- The **-P** switch, described in "Parallel Controls" on page 3-3, which specifies that **pmake** commands run in a compute partition.
- The **-u** switch, which causes **pmake** not to unlink files automatically checked out from SCCS or RCS. This switch can be useful when an error occurs where an intermediate source file must be made to make an object file. Use the **-u** switch to see the contents of the intermediate source file, rather than allowing **pmake** to remove it.

The **pmake** utility also provides the two special targets for specifying entry and exit code: **.INIT** and **.EXIT**. If you define **.INIT** in your *makefile*, this target and its dependencies are built before any other targets are processed. Defining **.EXIT** causes this target and its dependencies to be processed after all other targets are built.

Unlike many **make** facilities, GNU **make** automatically exports all variables (macros) from the *makefile* to the environment. In contrast, the **pmake** utility does not, but does allow you to export variables explicitly by using the special target **.EXPORT**. Variables listed as dependencies of this target are expanded and exported to the environment in which **pmake** runs its commands.

In addition to the standard GNU **make** include statement, **pmake** adds another **include** statement with the following form:

-include *filename*

The standard form of the **include** statement includes and processes the named file, and returns an error if the file is not found. The form of the include with the added dash prefix, includes and processes the named file as does the other version, but does not return an error if the file is not found.

The **pmake** utility adds some environment variables to those provided by GNU **make**. The syntax for all **pmake** switches, environment variables, macros, pseudotarget names, and conditional expressions, both extensions and those standard in GNU make, are listed in the **pmake** manual page (online or in the *Paragon™ OSF/1 Commands Reference Manual*).

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X