October 1993
Order Number: 312887-001

Paragon™ Graphics Libraries

User's Guide

Intel® Corporation

Revision	History	Date
-001	Original Issue	10/93

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

This manual tells how to use the graphics libraries available for use with the Paragon™ OSF/1 operating system on an Intel supercomputer.

This manual assumes that you are an application programmer proficient in the C or Fortran language, the UNIX operating system, and the graphics library in question. It provides you with the information you need to use the libraries effectively in parallel programs; however, it does not give general information on the graphics libraries. General information is provided by third-party manuals, which are discussed later and in each chapter.

Organization

Chapter 1	Tells you how to use the X Window System and the Athena widgets (which are provided with the Paragon OSF/1 base product), and the Motif widgets (an optional product).
Chapter 2	Tells you how to use DGL, the Distributed Graphics Library (an optional product).
Chapter 3	Tells you how to use OpenGL (an optional product).

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. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

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 dots) 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. See the *Paragon™ System Technical Documentation Guide* for information on the complete Paragon document set and ordering information.

Paragon™ Manuals

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

For information about limitations and workarounds, see the *Paragon™ System Software Release Notes for the Paragon™ XP/S System.* Release notes are also located in the directory */vol/share/release_notes* on your Paragon system.

Other Manuals

- *OSF/1 User's Guide*
- *OSF/1 Command Reference*
- *OSF/1 Programmer's Reference*
- *X Toolkit Intrinsics Programming Manual*
- *X Toolkit Intrinsics Reference Manual*
- *Xlib Programming Manual*
- *Xlib Reference Manual*
- *Motif Programming Manual*
- *X Toolkit Intrinsics Programming Manual — Motif Edition*

Comments and Assistance

Intel Supercomputer Systems Division is eager to hear of your experiences with our products. 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
Internet: 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)

Intel Japan 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
85622 Feldkirchen bei 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 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 629-9147

If you have comments about our manuals, please fill out and mail the enclosed Comment Card. You can also send your comments electronically to the following address:

techpubs@ssd.intel.com

Table of Contents

Chapter 1
Using the X Window System

Chapter 2
Using the Distributed Graphics Library

Chapter 3
Using the OpenGL Graphics System

List of Illustrations

List of Tables

Using the X Window System 1

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. Applications using the X Window System must be written in the C language.

A set of X Window System client libraries is included with Paragon™ OSF/1. The Motif libraries, a set of X Window System client libraries that implement the OSF Motif standard for "look and feel," are available as an option.

This chapter describes:

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

This chapter contains information specific to writing X applications for Paragon OSF/1 only. It does not describe how to write X Window System application 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(1)** online manual page, and "Authorizing the Supercomputer to Access the Server" on page 1-18.

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 1-17.

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 the program properly, as described in "Compiling and Linking X Window System Applications" on page 1-13.

A Sample X Program

To help you start using X in the compute partition, a sample program called *graph* is in the directory */usr/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.

A Motif version of the *graph* program, called *mgraph*, is available in */usr/share/examples/c/motif*. The Motif version has the same program logic, but uses the optional Motif widgets rather than the Athena widgets. The Motif widgets, which are not part of the standard Paragon OSF/1 product but are available as a separate option, can be used to give your program an industry-standard user interface. For information on Motif, refer to the *Motif Programming Manual* from O'Reilly and Associates.

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.

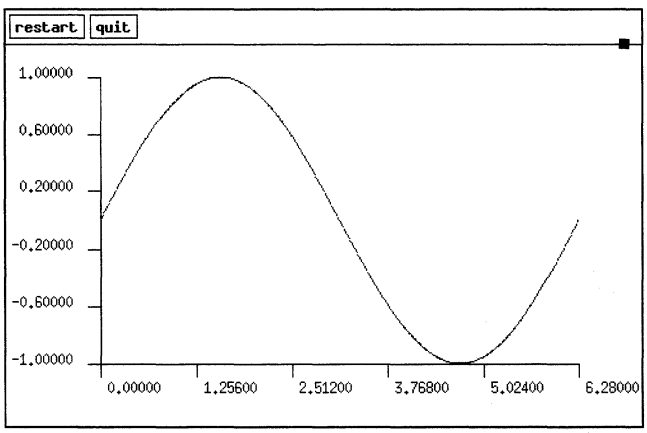

Figure 1-1. The *graph* Program Display

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 /usr/share/examples/c/xtoolkit/* 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:

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

```
mysun% rcp graph super:
mysun% 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:

```
mysun% 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 1-16.

Refer to the *Paragon™ 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.

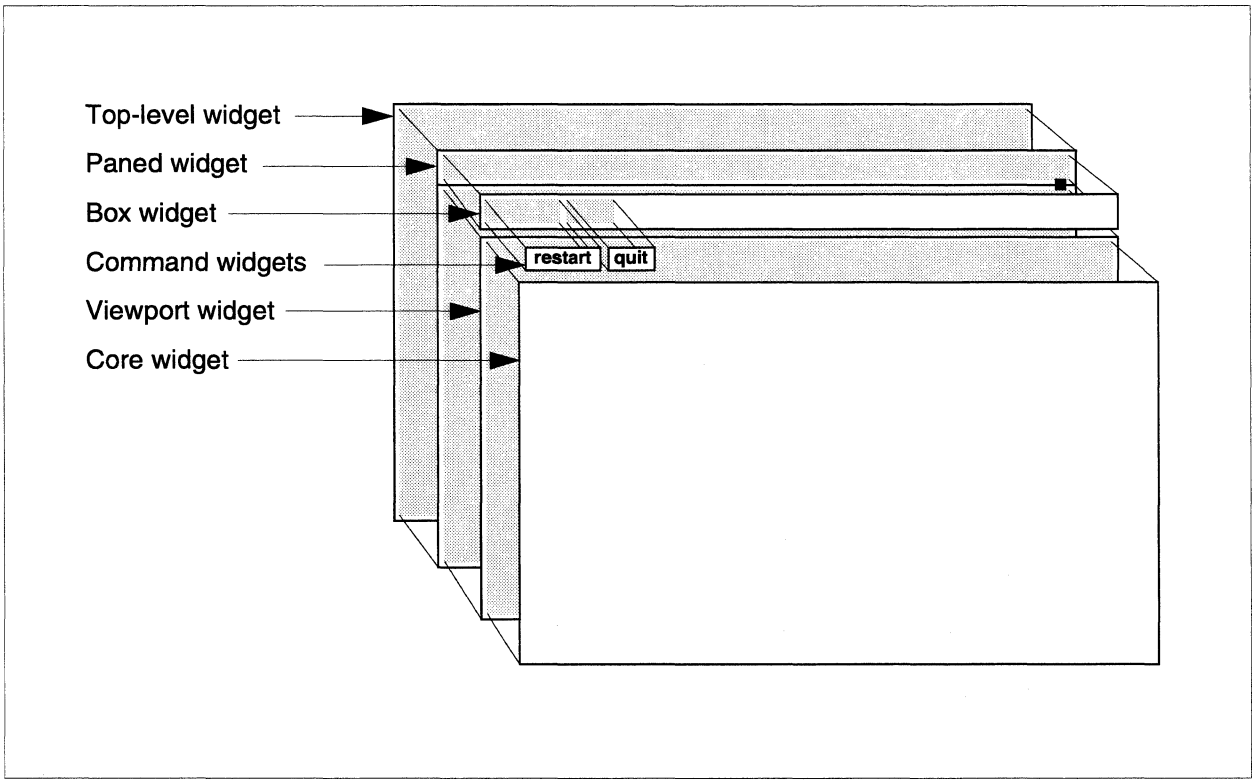

Figure 1-2. *graph* Program Widget Hierarchy

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 X events. This loop is often coded as either an infinite loop that calls **XNextEvent()** (which blocks until an event is received) or as a single call to **XtAppMainLoop()** (which never returns).

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. Thus, when no X events are coming in, the node cannot calculate or deal with messages from other nodes. One way to avoid this problem is to use an X Toolkit *work procedure* to handle messages.

Writing a Work Procedure

A work procedure is a user-supplied function that is called by the X Toolkit whenever it is idle waiting for an event. This function must return quickly to prevent the program's user interface response time from degrading; typically, response delays of more than a tenth of a second are considered unacceptable. However, a tenth of a second is enough time to receive a message and graph its contents.

A work procedure must be a function of type **Boolean**; its return status indicates whether or not it should be removed after it returns. A return status of **False** means that the work procedure should be called again the next time the Toolkit is idle, while a return status of **True** means that the work procedure has finished doing its job and does not need to be called again.

The *graph* program uses a work procedure called **HandleMessages()**, which looks like this:

```
/* global variables */
struct {
    int npoints;
    XPoint points[NPOINTS];
} points;       /* data points */
long msgid; /* message ID for incoming message */
        .
        .
        .
Boolean
HandleMessages()
{
    if(msgdone(msgid)) {
        XDrawPoints(XtDisplay(bitmap), picture, draw_gc,
                points.points, points.npoints,
                CoordModeOrigin);
        RedrawPicture(bitmap, NULL, NULL, NULL);

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

HandleMessages() uses the global variable *msgid*, which holds the message ID of an **irecv()**, and the global structure *points*, which holds the message when it is received. When this function is called, it tests to see if the **irecv()** has completed.

- If the receive has completed, the function graphs the values in the message in the internal bitmap called *picture*, copies *picture* to the screen by calling the user-provided function **RedrawPicture()** (which will be discussed later in this chapter), posts another **irecv()** to receive the next message, then returns **False**.
- If the receive has not completed, the function simply returns **False**.

Note that **HandleMessages()** uses the non-blocking call **irecv()** rather than a blocking **crecv()**. Making any blocking call within a work procedure can cause your program's window to appear to hang.

Installing a Work Procedure

You install a work procedure by calling **XtAppAddWorkProc()**. This call takes three arguments: the application context of the calling application, a pointer to the work procedure function, and an argument to be passed to the work procedure when it is called.

The *graph* program uses the following code to initialize *msgid*, install **HandleMessages()** as a work procedure, and begin the program's main event loop:

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

    /* arrange for messages to be handled while idle */
    (void)XtAppAddWorkProc(app_context,
                           (XtWorkProc)HandleMessages,
                           (XtPointer)NULL);

    /* infinite loop for X events and messages */
    XtAppMainLoop(app_context);
}
```

The call to **XtAppMainLoop()** never returns. However, because the program has registered **HandleMessages()** as a work procedure, it can handle messages whenever it is not busy dealing with X events.

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;

    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);
    }

    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.

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 compute partition, use the **icc** command with the **-nx** or **-lnx** switch and one or more of the **-l** switches shown in Table 1-1, Table 1-2, and Table 1-3. Other compiler switches may be used as well. For information on the switches that the compiler accepts, refer to the *Paragon™ C Compiler User's Guide*.

Standard X Window System Libraries

Nine X client libraries are included with Paragon OSF/1: 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	Description	Documentation	Link Switch
Xlib	Core X Window System library	*Xlib Programming Manual* *Xlib Reference Manual*	**-lX11**
Xaw	Athena widget set	*X Toolkit Intrinsics Programming Manual* *X Toolkit Intrinsics Reference Manual*	**-lXaw**
Xmu	MIT miscellaneous utilities	*Xlib Reference Manual*	**-lXmu**
Xt	Toolkit intrinsics layer	*X Toolkit Intrinsics Programming Manual* *X Toolkit Intrinsics Reference Manual*	**-lXt**
oldX	X10 compatibility library	*Xlib Reference Manual*	**-loldX**

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	Description	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**

Some of these 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*) for the compute partition, use a command line like the following:

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

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

Motif Libraries

The Motif widget set is an optional product. It is supplied in three libraries and documented in two volumes of the X Window System manuals by O'Reilly and Associates, as shown in Table 1-3.

Table 1-3. Motif Libraries

Library Name	Description	Documentation	Link Switch
Xm	Motif widget set	*Motif Programming Manual* and *X Toolkit Intrinsics Programming Manual — Motif Edition*	**-lXm**
Mrm	Motif resource manager utilities	*Motif Programming Manual*	**-lMrm**
Uil	User Interface Language compiler functions	*Motif Programming Manual*	**-lUil**

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

```
-lUil -lMrm -lXm -lXt -lXext -lX11 -lPW
```

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.

NOTE

The Motif libraries depend on the Programmer's Workbench library, **-lPW**. This library contains miscellaneous utility functions.

For example, to compile and link a program that uses the Motif widgets (but not the Motif resource manager utilities or UIL) for the compute partition, you could use a command line like the following:

```
% icc -nx filename -lXm -lXt -lXext -lX11 -lPW
```

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

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 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 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 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:

```
% 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 running a program that makes X client calls on the Intel supercomputer, issue the following command (if you use a shell other than **csh**, use the appropriate commands for your shell instead):

```
% 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 on the Intel supercomputer 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 this is not possible, you can use the server's Internet Protocol (IP) address instead of its name when specifying the display. For example:

```
% setenv DISPLAY 123.45.678.90:0
```

   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%
```

You can also use the server's Internet Protocol (IP) address instead of its name. For example:

```
mysun% xhost 123.45.678.91
```

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

Introduction

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 the 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 parallel DGL programs.
- How to compile and link DGL applications.
- What to do if your 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 application 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 the program properly, as described in "Compiling and Linking DGL Applications" on page 2-9.

A Sample DGL Program

To help you start using DGL in the compute partition, a sample program called *graph* is in the directories */usr/share/examples/fortran/dgl* and */usr/share/examples/c/dgl* on your Paragon system. (The versions in the two directories are identical in operation, but one is written in Fortran and the other in C.) The *graph* program demonstrates the special programming techniques that you can use to write DGL 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 a DGL 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 DGL program, or you may wish to examine the code for programming techniques.

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 pop-up menu, accessible by pressing the right-hand mouse button while the pointer is in the window, has two items: **Restart**, which clears the window and restarts the calculation from the beginning, and **Quit**, which terminates 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.

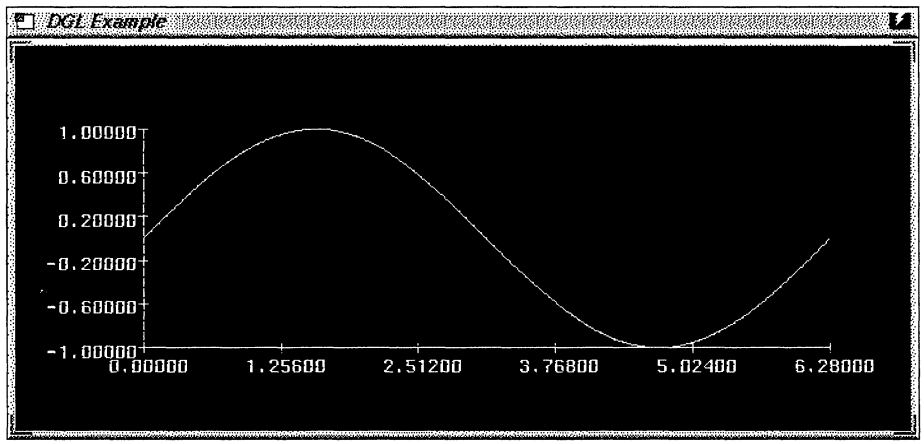

Figure 2-1. *graph* Program Display

Compiling, Linking, and Executing the *graph* Program

This section describes how to compile and link the C or Fortran language version 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.

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 one of the following commands:

```
% cp /usr/share/examples/fortran/dgl/* graph
        or
% cp /usr/share/examples/c/dgl/* 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:

```
mysgi% rcp "super:/usr/share/examples/fortran/dgl/*" graph
        or
mysgi% rcp "super:/usr/share/examples/c/dgl/*" 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:

```
mysgi% rcp graph super:
mysgi% 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 you see an error message instead, refer to "Problems Opening the Display" on page 2-10.

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

8. To draw the graph again, select **Restart** from the pop-up menu if you want to draw the graph again. To quit, select **Quit** from the menu.

Flow of Control in the *graph* Program

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

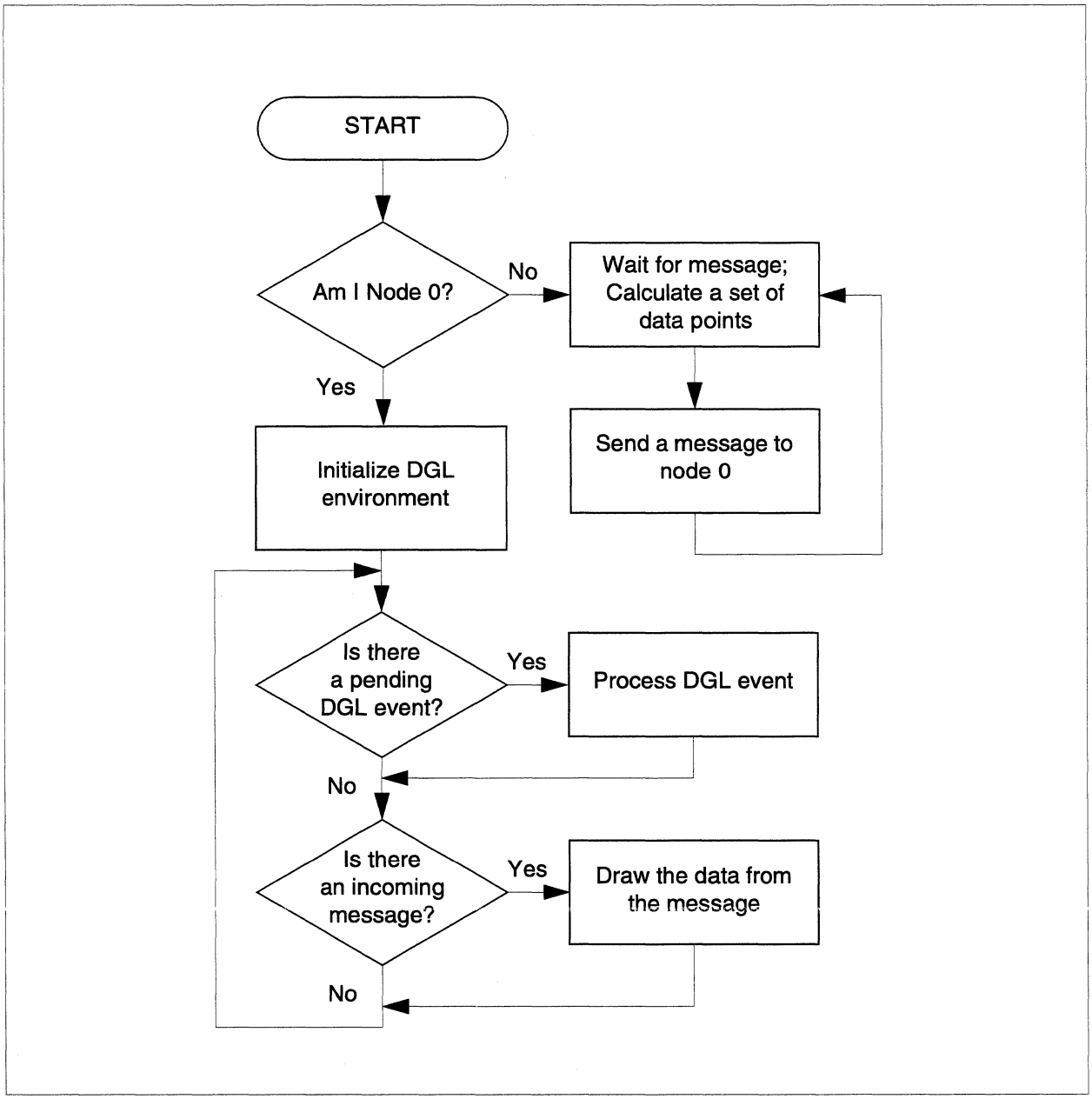

Figure 2-2. Flow Chart of the *graph* Program

Node 0 sets up the display window, pop-up menus, and other user interface elements, 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. To do this, you can 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 a call to **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 the compute partition, 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 in the compute partition. 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 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 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 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* or *REMOTEHOST* environment variable on the Intel supercomputer to the appropriate value for your server. If both variables are set, the value of *DGLSERVER* is used. You can use the following command on the Intel supercomputer 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 running a program that makes DGL client calls, issue the following command (if you use a shell other than **csh**, use the appropriate commands for your shell instead):

```
% 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 2-14.

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 name
libdgl error (write): value
```

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
PING super (012.34.567.891): 56 data bytes
64 bytes from 012.34.567.891: icmp_seq=0 ttl=255 time=10 ms
64 bytes from 012.34.567.891: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 012.34.567.891: icmp_seq=2 ttl=255 time=0 ms
<Del>

----super PING Statistics----
3 packets transmitted, 3 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 0/2/10
mysgi%
```

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

```
/usr/etc/ping: super: Unknown host
```

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

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

This error means that you are not able to **rlogin** from the Intel supercomputer to the SGI workstation without being prompted for a password. If you see this error, add the Intel supercomputer's name to the file *.rhosts* in your home directory on the SGI workstation. For example, if the name of the Intel supercomputer is *super*:

```
mysgi% cat >> ~/.rhosts
super
```

```
<Ctrl-d>
mysgi%
```

Once you have done this, you should be able to log into the SGI workstation from the Intel supercomputer without having to supply a password. If this is not the case, see your system administrator for assistance.

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 *hostname*, 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

Using the OpenGL Graphics System 3

Introduction

The OpenGL graphics system is a hardware-independent library of graphical commands. You use OpenGL library calls in your program (an OpenGL *client*), to create images of three-dimensional objects on a display attached to a separate computer (an OpenGL *server*). OpenGL provides colors, textures, materials, lighting, shading, and atmospheric effects that you can use to produce photorealistic images.

However, OpenGL is not a window system: it is only a system for drawing. OpenGL must be used in conjunction with a window system, such as the X Window System (described in Chapter 1). The window system is used to control windows, buttons, scroll bars, and other user interface elements, and OpenGL is used to control the rendering of images within a window.

This chapter describes:

- How OpenGL is implemented in Paragon™ OSF/1.
- Where to look for documentation on OpenGL.
- How to link OpenGL programs.

This chapter contains information specific to OpenGL in Paragon OSF/1 only. It does not describe how to write OpenGL programs.

OpenGL in Paragon™ OSF/1

OpenGL in Paragon OSF/1 is an extra-cost optional product (the OpenGL software is provided on the same tape as the optional DGL software described in Chapter 2).

The Paragon OSF/1 implementation of OpenGL is a library of X Window System calls. These calls can only be used with an X server that supports the OpenGL extension (you can use the call **glXQueryExtension()** in your OpenGL programs to determine whether or not the currently-connected server supports this extension). The server is not provided with Paragon OSF/1; it must be obtained from a third party.

Applications using OpenGL must be written in the C language. OpenGL programs can run either in the service partition or the compute partition.

NOTE

Because OpenGL is provided as an extension to X, all the techniques described for writing and executing parallel X programs in Chapter 1 apply to OpenGL as well.

For example, if your OpenGL program has problems opening a connection with the server, see "Problems in Opening the Display" on page 1-16.

For an introduction to using OpenGL with X, see the manpage **glXIntro(3)**.

OpenGL Documentation

Online manual pages for all the calls in the OpenGL libraries are provided on your Paragon XP/S system. For printed documentation and tutorials, see the OpenGL documentation provided with your server. The following books, which discuss OpenGL in a server-independent fashion, are also available at many technical bookstores:

- *OpenGL Programming Guide*, ISBN 0-201-63274-8.
- *OpenGL Reference Manual*, ISBN 0-201-63276-4.

Both these books were written by the OpenGL Architecture Review Board (Jackie Neider, Tom Davis, and Mason Woo) and published by Addison-Wesley Publishing Company.

Linking OpenGL Programs

When linking an OpenGL program, use the following switches (in the following order):

```
[ -lGLU ] -lGL [ other X libraries ] -lXext -lX11
```

The **-lGL** switch links to the OpenGL library (calls whose names begin with **gl** or **glx**); the optional **-lGLU** switch links to the OpenGL Utility library (calls whose names begin with **glu**). Other compiler switches may be used as well. For information on the switches that the compiler accepts, refer to the *Paragon™ C Compiler User's Guide*.

For example, to compile and link an OpenGL program that uses the Athena widgets (**-lXaw -lXmu**) for execution in the compute partition (**-nx**), use a command line like the following:

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

To compile the same program for execution in the service partition, use the same command line without the **-nx** switch.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X