

R.V. COLLEGE OF ENGINEERING
BANGALORE

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

NETWORK PROGRAMMING LAB
PROJECT REPORT

**PARALLEL COMPUTING USING
LINUX CLUSTERS**

PFRACT – A PARALLELIZED FRACTAL
GENERATION PROGRAM

DEVELOPED BY

SRIVAS N. CHENNU 1RV98CS086

VISHWAS N. 1RV98CS104

**R.V. COLLEGE OF ENGINEERING
BANGALORE**

**DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING**

CERTIFICATE

This is to certify that

Srivas N. Chennu (1RV98CS086)

studying in 7th semester Comp. Sc. and Engg. has satisfactorily
completed the project work on

Parallel Computing using Linux Clusters

in fulfillment of the Network Programming Lab syllabus, as
prescribed by the VTU for the academic year 2001-2002.

INTERNAL EXAMINER

EXTERNAL EXAMINER

PROF. B. I. KHODANPUR

H.O.D. CSE Dept.

TABLE OF CONTENTS

1.	Introduction.....	5
	Cluster Computing Using Linux	6
	The Parallelization Process	8
	Introduction To Fractals	9
2.	Software Requirements Specification	10
3.	Synopsis.....	11
4.	Design	13
4.1.	Layered Architecture.....	14
	The Parallelization Engine	14
	The Communication Manager.....	16
	The User Interface	21
4.2.	Design Components.....	24
	Synchronization Issues	25
5.	Implementation.....	26
5.1.	Dependencies	26
5.2.	Source Code.....	32
5.2.1.	The Pfract Client.....	33
	Pfract.H.....	33

Pfract.Cpp.....	34
Drawview.H	41
Drawview.Cpp	43
Clusternode.H.....	63
Clusternode.Cpp.....	63
Main.Cpp	65
5.2.2. The Pfract Server.....	67
Fract.H	67
Fract.Cpp	67
Mandel.H	69
Mandel.Cpp.....	70
Julia.H.....	71
Julia.Cpp.....	73
Pfractd.H.....	74
Pfractd.Cpp	75
Main.Cpp	77
6. Testing.....	80
7. Conclusion.....	80
The End	80

1. INTRODUCTION

Parallel computing is a concept that involves exploiting the parallelism in computer programs. The major goal is to improve the performance of computer systems by utilizing their processing capabilities so as to gain maximum throughput.

Some of the important advantages of parallel computing are -

- ✓ Availability of low cost electronic processing hardware.
- ✓ Availability of high-speed network architectures.
- ✓ Dramatic speedup in processing of complex computing problems.

Parallel computing has applications in a variety of fields. Some of these are listed below.

- ✓ Scientific computing – Large parallel systems are commonly used to solve many problems that occur in the areas of Physics, Chemistry, Biology, Weather Forecasting etc.
- ✓ Commercial computing
- ✓ Engineering applications – Engineering fields like Mechanical, Civil and Aeronautical engineering apply parallel systems in CAD/CAM design.

CLUSTER COMPUTING USING LINUX

Hardware architectures for Parallel Computing - There are basically two ways parallel computer hardware is put together:

- ✓ Local memory machines that communicate by messages (Clusters)
- ✓ Shared memory machines that communicate through memory (SMP machines)

A typical Linux cluster is a collection of single CPU machines connected using a fast Ethernet running the Linux operating system, and a accompanying parallel processing software engine. On the other hand, A Symmetric Multiprocessing (SMP) architecture is a shared memory machine, with a set of processors computing in parallel, operating on a shared memory area.

The local memory architecture can be scaled up to large numbers of CPUs, while the number of CPUs shared memory machines can have is limited due to memory contention problems. It is possible to connect many shared memory machines to create a hybrid shared memory machine. These hybrid machines look like a single large SMP machine to the user and are often called NUMA (non uniform memory access) machines because the global memory seen by the programmer and shared by all the CPUs can have different latencies. At some level, however, a NUMA machine must pass messages between local shared memory pools. It is also possible to connect SMP machines as local memory computing nodes.

Cluster computing in Linux involves a collection of Linux boxes connected by a network, working in parallel in order to speed up the process of finding solutions to large, complex parallel computing problems. A Linux cluster typically consists of a single master node that controls all the other slave nodes. The master initializes all the slave nodes, and sends the computational tasks, which are computed at the nodes. The master then collects the results from the slave nodes and merges them into the resulting output.

Linux clusters offer a highly scalable parallel computing architecture. Computing power in the individual nodes can be kept at moderate levels. Parallel software architectures operate on the cluster hardware and provide a platform for a programmer to split up a program into parallelized tasks and efficiently distribute them among the machines in the cluster. Additional computing resources can be added to the cluster as and when required, with minimal changes to the existing system. These features make Linux Clusters ideal for large scale parallel computing.

THE PARALLELIZATION PROCESS

Parallelization is the process of identifying and distributing the inherently parallel sections of a program to individual processing elements in a parallel architecture. Additionally, the results computed by the individual nodes have to be collected and merged to produce the final output. The important steps involved in parallelizing an algorithm are as below.

1. **Decomposition** – In this step, the aim is to expose the inherent concurrency in the program. Such concurrent sections ideally do not have any interdependencies, and thus can be executed in parallel to achieve speedup.
2. **Assignment** – Tasks are assigned to individual processing elements such that:
 - a. Workload is balanced.
 - b. Communication volume is kept at a minimum.
 - c. Optimum throughput is obtained.
3. **Orchestration** – Orchestration involves the need to synchronize the operations at the processing elements so that data and control dependencies are satisfied. The aim here is to keep serialization of tasks to a minimum, at the same time, satisfying dependencies between components.
4. **Mapping** – Mapping is the final step where tasks are allotted for execution at processing nodes. This involves choosing a configuration such that related tasks are scheduled on the same processor, and locality in the network topology is exploited.

INTRODUCTION TO FRACTALS

Fractals are graphical shapes that are produced by the iterative application of a specific equation to points in an n -dimensional space. This special type of equation is typically a feedback equation, in which the result of iteration is used as the input to the next. Fractal images display properties of self-similarity and infinite boundary lengths. This means that sections of the fractal image can be magnified repeatedly to see structures that are very similar to the parent fractal. Additionally the measurement of the exact length of a boundary of a fractal is impossible.

Fractals are called so because they have non-integral dimensions. Fractal structures have many instances in nature. Typical examples of fractal structures occurring in natural systems are fern leaves, Lorenz attractors etc.

The Mandelbrot and Julia sets are some of the commonly known synthetic fractal images. These fractals can be generated by means of a simple iterative equation that identifies a set of points that belong to the fractal image. The iterations are applied to each point in an n -dimensional space repeatedly. Depending on the results generated by the iterations, the point is either included or excluded from the set. The numbers of iterations that are executed for each point determine the depth and clarity of the resulting fractal image.

The computation of fractal images is a processor intensive task. Hence, it is a good candidate for parallel computation. Additionally, it lends itself well to parallelization, due to the concurrency inherent in the algorithm used to generate fractals. The computation of any fractal point is independent of any other point. Hence, the iterative application of the fractal equation to individual points in the image can be done in parallel at different nodes in the cluster. This greatly increases the speed of generation of the images, even when a large number of iterations are being executed. Parallelization thus enables generation of fractals of high clarity and depth in reasonable amount of time.

2. SOFTWARE REQUIREMENTS SPECIFICATION

Listed below is the requirements specification for PFract. It lists the basic requirements to be satisfied by the design of the PFract architecture.

- ❑ It should be implemented for the Linux Operating System.
- ❑ It should employ the Linux cluster architecture for parallel computation.
- ❑ It should make optimum use of the processing nodes.
- ❑ It should make optimum use of the network resources.
- ❑ The network load should be monitored and kept at a minimum.
- ❑ It should implement a generic parallelization engine that can be easily adapted for other parallel applications.
- ❑ It should incorporate an implementation-independent design.
- ❑ It should render fractal images in high depth and quality
- ❑ It should provide an intuitive and well-designed graphical user interface.
- ❑ It should allow the user to control the parameters of the generated fractal images.
- ❑ It should allow the user to easily manipulate the fractal images.
- ❑ It should provide high performance with respect to speed of fractal generation.

3. SYNOPSIS

PFract – short for Parallel Fractal Generator is a parallel computing initiative using Linux clusters. PFract is a parallel processing application that employs a generic parallelization engine to draw complex fractal images. The computations involved in generating the fractal images are executed in parallel on all the nodes in the Linux cluster. PFract accomplishes the following operations –

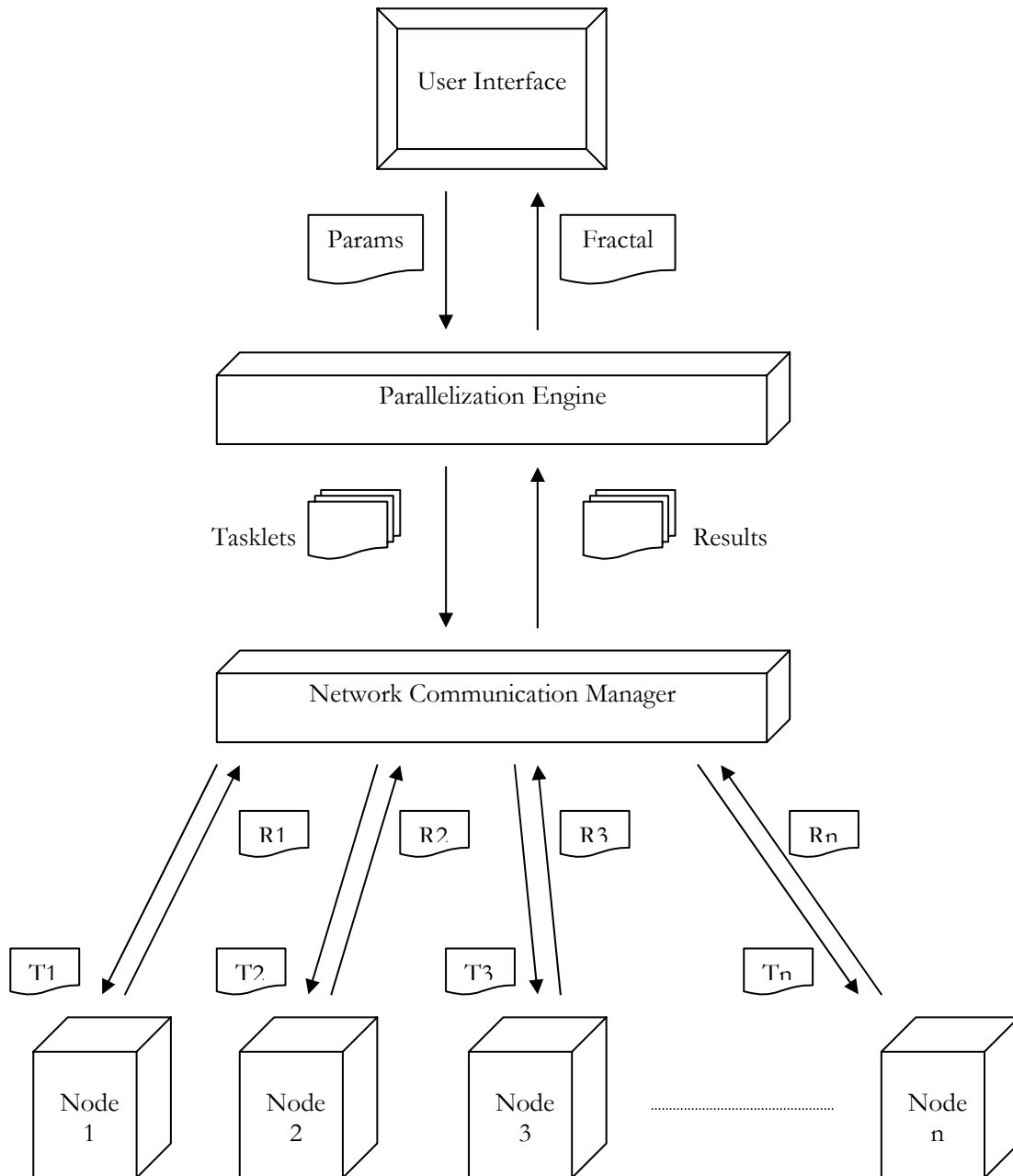
- **Parallelization** – The generic computing engine deployed in PFract parallelizes the task of computing the fractal image points by splitting up the processing into optimum tasklets.
- **Deployment** – The engine sends each tasklet to a processing node for computation.
- **Synchronization** – Processing of tasks at the nodes and communication mechanism within the cluster is synchronized in order to maintain data and control consistency.
- **Result Collection** – Once the processing nodes have processed their assigned tasklets, the engine collects results from the nodes and aggregates them at the master node.
- **Formatting and Display** – The data computed by the nodes are finally displayed to the user in the form of a fractal image.

Some of the important features of PFract are given below –

- ❑ **Master-Slave Architecture** – PFract extends the generic client-server architecture, by employing a single client or master, and multiple servers or slaves, which are controlled by the master.
- ❑ **Object Oriented Design** – The core parallelization engine of PFract is object oriented in design, at both client and the server.
- ❑ **Generic Implementation** – The PFract parallelization engine is generic in design and can easily be adapted for any other parallel computing application.
- ❑ **Flexible Graphical User Interface** – PFract can display richly colored fractal images and offers the user a full fledged interface to configure a variety options for fractal generation, with regard to depth, position etc.

4. DESIGN

The architecture of the PFract system is illustrated in the block diagram below.



PFract Master Slave Architecture

4.1. LAYERED ARCHITECTURE

The design of PFract can be broadly divided into the following distinct layers. These layers, as illustrated by the diagram above, are detailed below.

THE PARALLELIZATION ENGINE

The core parallelization engine of PFract has a generic objected-orient design. Before detailing the architecture of the engine, the basic fractal generation algorithm for the Mandelbrot set is detailed below.

Step 1 – Initialize the fractal plane as 2-Dimensional plane of complex numbers of the form $(X + iY)$.

Step 2 – Consider a point $(X + iY)$ in the fractal plane. Initialize $C = X + iY$.

Step 3 – Iteratively compute the feedback equation $Z_{n+1} = Z_n^2 + C$ where $Z_0 = 0$.

Step 4 – If the value of Z rapidly increases towards infinity, the point $(X + iY)$ does not belong the fractal set. In this case, the Z value moves further and further away from the center of the plane. On the other hand, if the value of Z converges to a single point or a finite set of points, it belongs to the set.

Step 5 – Mark the point in the fractal plane, using a color-coding scheme that corresponds to the iteration count for the point.

Step 6 – Repeat above steps for every point in the current fractal region.

The parallelized version of the above algorithm employed by PFrac is given below. It parallelizes the concurrent parts of the algorithm, executes them in the processing nodes of the cluster, and collects the results for display.

Step 1 – Initialize the fractal plane as 2-Dimensional plane of complex numbers of the form $(X + iY)$.

Step 2 - Let N be the set of nodes in the cluster having a total of C computing nodes. Initialize a Boolean array $Used[C]$ of FALSE values.

Step 3 – Consider a line in the complex plane with the equation $Y = Y_i$.

Step 4 – Select a Node N_j such that $Used[j] = FALSE$. Send the Y_i coordinate of the line to a processing node N_j . Set $Used[j] = TRUE$.

Step 5 – In the node N_j , compute the iteration counts for all the points lying on the line. Return the computed values to the master node.

Step 6 – If no unused node (with $Used[j] = FALSE$) can be found, obtain computed results on a First-Come-First-Serve basis from the computing nodes.

Step 7 – If node N_j has finished computing, store the results and set $Used[j] = FALSE$.

Step 8 – Display the obtained results using a color-coding scheme that corresponds to the iteration count for the points.

Step 9 – Repeat the above steps for every line in the current fractal region.

THE COMMUNICATION MANAGER

The Network communication manager in PFract uses bi-directional TCP sockets to accomplish communication within the cluster of nodes. A brief introduction to the Linux TCP and Sockets implementation is given below.

TCP

This is an implementation of the TCP protocol as defined in RFC793, RFC1122 and RFC2001. It provides a reliable, stream oriented, full duplex connection between two sockets on top of the Internet Protocol (IP). TCP guarantees that the data arrives in order and retransmits lost packets. It generates and checks a per packet checksum to catch transmission errors. TCP does not preserve record boundaries. A fresh TCP socket has no remote or local address and is not fully specified. To create an outgoing TCP connection, **connect** is used to establish a connection to another TCP socket. To receive new incoming connections **bind** is used to give the socket a local address and port. Then **listen** is called to put the socket into listening state. After this, a new socket for each incoming connection can be accepted using **accept**.

A socket which has had **accept** or **connect** successfully called on it is fully specified and may transmit data. Data cannot be transmitted on listening or not yet connected sockets. TCP is built on top of IP. The address formats defined by IP also apply to TCP. TCP only supports point-to-point communication; broadcasting and multicasting are not supported.

UDP

This is an implementation of the User Datagram Protocol described in RFC768. It implements a connectionless, unreliable datagram packet service. Packets may be reordered

or duplicated before they arrive. UDP generates and checks checksums to catch transmission errors. When a UDP socket is created, its local and remote addresses are unspecified.

Datagrams can be sent immediately using **sendto** or **sendmsg** with a valid destination address as an argument. When **connect** is called on the socket the default destination address is set and datagrams can now be sent using **send** or **write** without specifying a destination address.

It is still possible to send to other destinations by passing an address to **sendto** or **sendmsg**. In order to receive packets the socket can be bound to a local address first by using **bind**. Otherwise the socket layer will automatically assign a free local port and bind the socket to any available address.

All receive operations return only one packet. When the packet is smaller than the passed buffer only that much data is returned, when it is bigger the packet is truncated. IP options may be sent or received using the socket options. UDP uses the IPv4 address format.

SOCKETS

A Socket creates an endpoint for communication and returns a descriptor. The Domain parameter specifies a communication domain; this selects the protocol family that will be used for communication.

The different types of socket protocols include -

UNIX	Local communication
INET	IPv4 Internet protocols
INET6	IPv6 Internet protocols
IPX	IPX - Novell protocols

NETLINK	Kernel user interface device
X25	ITU-T X.25/ISO-8208
AX25	Amateur radio AX.25
ATMPVC	Access to raw ATM PVCs
APPLETALK	Appletalk
PACKET	Low-level packet interface

The socket has an indicated type, which specifies the communication semantics. The types of sockets that can be created are -

STREAM

Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

DATAGRAM

Supports datagrams, which are connectionless, unreliable messages of a fixed maximum length.

SEQUENCED PACKET

Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.

RAW

Provides raw network protocol access.

RELIABLE DATAGRAM

Provides a reliable datagram layer that does not guarantee ordering.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the communication domain in which communication is to take place.

Stream Sockets are full-duplex byte streams, similar to pipes. They do not preserve record boundaries. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a **connect**. Once connected, data may be transferred using **read** and **write** calls or some variant of the **send** and **recv** calls. When a session has been completed a close may be performed. **Out-of-band** data may also be transmitted using **send**. and received using **recv**.

The communications protocols that implement a stream socket ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered to be dead.

A variety of library routines are provided to supplement the basic socket library. These libraries provide the application programmer with the following functionality –

- ✓ Conversion between machine-readable and human-readable Internet addresses.
- ✓ Address lookup using the DNS.
- ✓ Manipulation of socket operation at various levels, using IOCTL and SYSCTL.
- ✓ Information about socket end point processes.

Sequenced Packet sockets employ the same system calls as stream sockets. The only difference is that **read** calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

Datagram and Raw sockets allow sending of datagrams to correspondents named in **send** calls. Datagrams are generally received with **recvfrom**, which returns the next datagram with its return address.

The UNIX I/O control libraries provide for two types of IO to be performed via sockets.

- ▶ **Synchronous mode** – The application manually polls the socket buffers to check whether data can be sent and whether new data has arrived.
- ▶ **Asynchronous mode** – The application can also designate a signal handling function and specify a signal that it should be sent when I/O is possible on the socket. This allows the application to continue normal processing instead of polling and/or waiting for data.

When the network signals an error condition to the protocol module the pending error flag is set for the socket. The next operation on this socket will return the error code of the pending error.

The communication manager maintains a bi-directional socket open to each node. Commands and computed data are sent on the same channel. Additionally, PFract provides a generic object oriented interface to the underlying connection implementation. This allows the Parallelization engine to transparently access the communication layer without knowing about the details of the implementation. This design feature allows the generic engine to be easily ported to any other implementation.

THE USER INTERFACE

PFract provides a friendly GUI based interaction interface to the user. This interface has been implemented using the Qt and KDE GUI programming libraries.

QT

Qt is a multi-platform, full-featured GUI design toolkit. Qt-based applications represent themselves with buttons, windows etc allowing user input by visualizing the functions an application provides. Such a toolkit is needed for developing graphical applications that run on the X-Window interface on Unix Systems, because X does not contain a pre-defined user interface itself.

Although other toolkits are also available to create User Interfaces, Qt offers some technical advantages that make application design very easy. Additionally, the Qt toolkit is also available for Microsoft Windows systems, which allows developers to provide their applications for both platforms.

The Qt library is a toolkit that offers graphical elements that are used for creating GUI applications and are needed for X-Window programming. Additionally, the toolkit offers:

- A complete set of classes and methods ready to use even for non-graphical programming issues.
- A good solution towards user interaction by virtual methods and the signal/slot mechanism.
- A set of predefined GUI-elements, called "widgets", that can be used easily for creating the visible elements
- Completely pre-defined dialogs that are often used in applications such as progress and file dialogs.

KDE

The K Desktop Environment offers an easy way for application designers to equip their products with an intuitive way of user interaction. It provides all means to solve common tasks using a set of library classes that extend the facilities of the Qt toolkit. This allows for a unique look and feel to applications as well as interaction with other programs and the window manager.

Basically, KDE offers a set of standards that allow a unique look and usage of applications that should be watched when designing programs. A lot of tasks are done automatically, such as:

- Session Management
- Standard keyboard accelerator configuration
- Font, Color and Style changing
- Theme support
- Internationalization

Therefore, these issues only have to be mentioned in their functionality for complete information. Application developers only have to care about what their program is intended to do and where KDE can help. There, KDE offers user interfaces that extend the Qt toolkit where necessary. If both libraries offer similar solutions, KDE developers should (in most cases) use the methods provided by the KDE libraries.

KDE offers a set of widgets that can be used for creating application specific dialogs and views. Examples are **KSeparator**, offering a common separator line, **KColorButton**, offering a push button displaying a color etc.

Normally, applications ask the user to select various values. Here, the libraries provide easy means to get these values by complex widgets that are ready to use and are already known to the user by the KDE desktop, such as:

- File dialogs
- Color dialogs
- Font dialogs
- Keyboard configuration dialogs

These should be used wherever a user setting is required as it simplifies the programmer's work, extends the application's facilities dramatically and provide a common look.

Also, KDE programs integrate themselves into the desktop and are able to interact with the file manager via drag and drop, offer session management and many more, if all features offered by the KDE libraries are used. Both, the Qt toolkit and the KDE libraries are implemented in the C++ programming language; therefore applications that make use of these libraries are also mostly written in C++.

PFract utilizes the Qt and KDE class libraries to create and manage its user interface. Results generated by the computing nodes are finally represented in the form of a graphic fractal image. The Qt libraries provide graphics rendering functionality at two distinct levels of abstraction.

- High-level graphic rendering via the **QpaintDevice**, **Qcolor** and **Qpainter** classes.
- Low-level graphics manipulation via the **bitBlit()** library functions.

4.2. DESIGN COMPONENTS

The PFract system can be divided into following components, with regard to the master – slave architecture.

THE PFRACT SERVER

The PFract Server process runs on every slave node in the cluster, and listens for client connections on a pre-designated port. Once it receives a connection, it waits for instructions from the client, in the form of messages. The PFract client communicates with the server by sending it a **message header**, followed by the **message data**, whose interpretation depends on the message type. The following types of message headers are defined in PFract.

- **MFracType** – This header instructs the server to set the current fractal type, to that specified in the **message data**.
- **MFracInit** – This header instructs the server to initialize a new fractal object of the predetermined type. The **message data** contains the initialization parameters.
- **MFracCalc** – This header instructs the server to calculate the iteration counts for a line ($Y = Y_i$). The **message data** contains the Y_i value to be used.
- **MDone** – This header tells the server that the client has finished computation and is exiting.

Depending on the type of header, the server performs the appropriate task and sends a reply back to the client.

THE PFRACT CLIENT

The PFract client process runs on the master node of the cluster. It has the following responsibilities to be fulfilled –

- ❑ Network communication management.
- ❑ Synchronization of task processing
- ❑ Result collection
- ❑ User interface management

When the PFract client initializes, it connects to all the active slave nodes in the cluster, and establishes a bi-directional TCP socket to each PFract server. Once connected, it invokes the parallelization engine to divide the fractal generation task into tasklets. After initializing the PFract servers, it sends the tasklets to the servers, and manages the results collections, as in the parallelization engine algorithm detailed above. Finally, it displays the results to the user by invoking the Qt graphics library to draw the fractal image onto the user's screen.

SYNCHRONIZATION ISSUES

The PFract client manages the synchronization issues that arise during the distributed computation of the task in the cluster. Firstly, data that is being sent to a server has to be kept track of, and when the results are obtained, correspondence between the request and reply has to be maintained. Since the PFract client obtains results from the servers on a FCFS basis, these results have to be ordered accordingly, before they can be sent for display.

Secondly, data transfer via the socket interface has to be reliable and sequenced, so as to ensure correct fractal image generation. The PFract client thus manages the socket I/O buffers to ensure that data is not lost and is not received out of order.

5. IMPLEMENTATION

The implementation features of the PFract system are listed below.

- ❑ PFract has been implemented using the C++ programming language.
- ❑ It is completely object oriented in its design and implementation.
- ❑ It has been implemented for operating systems which comply with the POSIX 1.1 standard.

5.1. SOFTWARE DEPENDENCIES

The implementation dependencies of PFract are detailed below.

The Communication Manager in the PFract system uses the Berkeley sockets library provided in the Linux Operating System for communicating between the client and the servers. A brief overview of the Berkeley sockets library is given below.

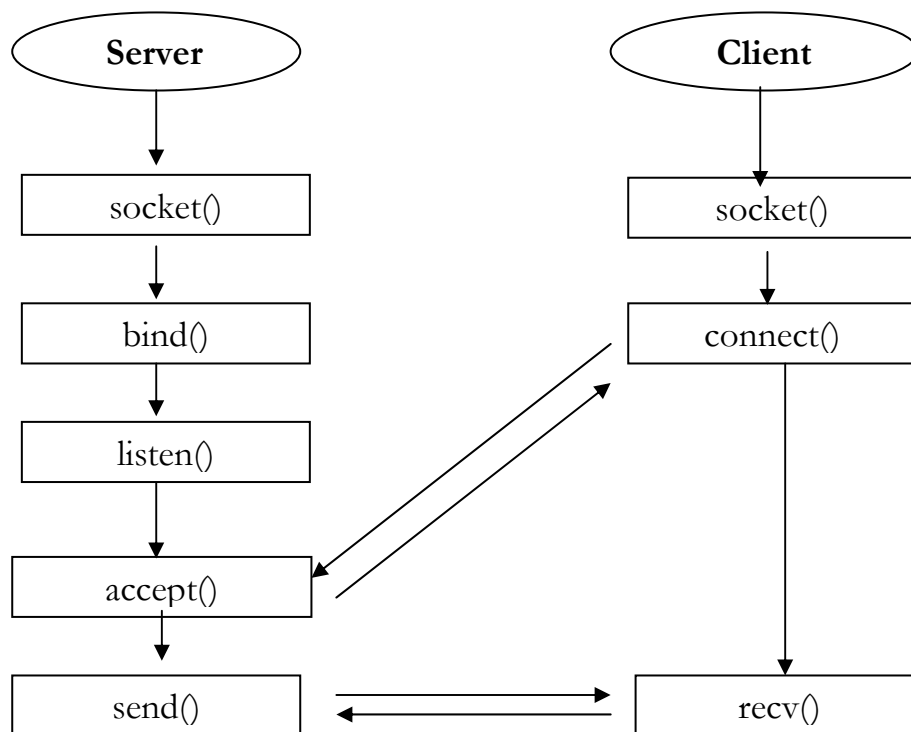
BERKELEY SOCKETS

A socket is a type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a port, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To

manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Connection Oriented Service - In a connection oriented service, the client and server establish a point-to-point connection, before transferring any data. Once the connection setup procedure is completed, a Virtual Circuit is established between the client and the server, whose ID is implicitly used in all communication between them. Once the data transfer is completed, the connection must be explicitly closed. Connection – oriented service using TCP in the Berkeley Sockets library guarantees reliable and sequenced packet delivery. This is the type of service used by the PFrac system.

For a client-server using a connection oriented protocol, the Berkeley socket system calls are as shown in the below figure.



Socket Addresses - Many of the BSD networking system calls require a pointer to a socket address as an argument. The definition of this structure is in `<sys/socket.h>`:

```
struct sockaddr    {
    u_short sa_family; /* Address family : AF_xxx value */
    char  sa_data[14]; /* Up to 14 bytes of protocol-specific addresses */
};
```

The contents of the 14 bytes of protocol specific addresses are interpreted according to the type of address. For the Internet family, the following structures are defined in `<netinet/in.h>`:

```
struct in_addr {
    u_long s_addr; /* 32-bit netid/hostid network byte ordered */
};

struct sockaddr_in {
    short  sin_family; /* AF_INET */
    u_short sin_port; /* 16 bit port number */
    struct in_addr sin_addr; /* 32 bit netid/hostid network byte ordered */
    char sin_zero[8]; /* unused */
};
```

The header file `<sys/types.h>` provides C definitions and datatype definitions (typedefs) that are used throughout the system.

Socket System Calls Used - In this section, we describe the elementary system calls required to perform network programming.

1. *socket* System Call:

To do network I/O, the first thing a process must do is call the socket system call, specifying the type of communication protocol desired (Internet TCP, Internet UDP, etc.)

```
# include <sys/types.h>
```

```
# include <sys/socket.h>
```

```
int socket (int family , int type , int protocol );
```

family - Specifies the protocol to be used.

Type - Specifies the socket type to be used.

Protocol - Specifies a particular protocol.

2. *bind* System Call:

The *bind* system call assigns a name to an unnamed socket.

```
# include <sys/types.h>
```

```
# include <sys/socket.h>
```

```
int bind (int sockfd , struct sockaddr *myaddr , int addrlen);
```

sockfd - is the socket file descriptor

myaddr - is a pointer a protocol-specific address

addrlen - is the size of the address structure.

Servers register their well known address with the system. Both connection oriented and connectionless servers need to do this before accepting client requests. A client can register a specific address for itself. The *bind* system call fills in the *local-addr* and local-process elements of the association 5-tuple.

3. *send* System Call:

The *send* system call is used to send data over a connected socket.

```
# include <sys/types.h>
```

```
# include <sys/socket.h>
```

```
int send (int sockfd , char *buff , int nbytes , int flags);
```

sockfd - is the socket file descriptor

buff - is a temporary buffer

nbytes - is th length of path of the filename

flags - is formed by ORing one of the following constants

MSG_OOB - send or receive out of band data

MSG_PEEK - peek at incoming message

MSG_DONTROUTE - bypass routing

4. *recv* System Call:

The *recv* system call is used to receive data from a connected socket.

```
# include <sys/types.h>
```

```
# include <sys/socket.h>
```

```
int recv ( int sockfd , char *buff , int nbytes , int flags , struct sockaddr *from , int
addrlen);
```

sockfd - is the socket file descriptor

buff - is a temporary buffer

nbytes - is the length of path of the filename

flags - is formed by ORing one of the following constants

MSG_OOB - send or receive out of band data

MSG_PEEK - peek at incoming message

MSG_DONTROUTE - bypass routing

THE QT AND KDE CLASS LIBRARIES

PFract either inherits or utilizes the functionality provided by the following classes provided by the Qt and KDE class libraries.

- ❑ Kapplication, Kmainwindow, Kaction, Kapp, KimageIO KfileDialog.
- ❑ Qdialog, Qvector, Qwidget, Qapplication
- ❑ Qpainter, Qpaintdevice, Qpen.

5.2. SOURCE CODE

The core class libraries of Pfract are given below, along with a brief description of their functionality.

- ❑ PFract – This is the main application class. It inherits from the KDE Kapplication class, and provides the basic GUI interface.
- ❑ DrawView – This class handles the details of drawing the fractal image to the screen. It has private class members which implement the graphics rendering functionality.
- ❑ Fract – This is the abstract base class from which the actual fractal classes inherit.
- ❑ IterDlg – The iteration count selection dialog box class. It inherits from QDialog.
- ❑ Julia – The Julia set class. It inherits from Fract.
- ❑ JuliaDlg – The Julia dialog box class. It inherits from QDialog.
- ❑ MandelDlg – The Mandelbrot dialog box class. It inherits from QDialog.
- ❑ Mandel – The Mandelbrot set class. It inherits from Fract.
- ❑ TypeDlg – The fractal type selection dialog box class. It inherits from QDialog.
- ❑ ZoomDlg – The zoom ratio selection dialog box class. It inherits from QDialog.
- ❑ ClusterNode – The communication manager class.
- ❑ PfractD – The PFract server class used at the slave nodes.

5.2.1. THE PFRACT CLIENT

PFRACT.H

```

#ifndef KFRACT_H
# define KFRACT_H

#include <qpainter.h>
#include <kapp.h>
#include <kmainwindow.h>
#include <kaccel.h>
#include <kkeydialog.h>
#include <kaction.h>

#include "drawview.h"
#include "kfractdlgs.h"
#include "version.h"

class FractMainWindow : public KMainWindow
{
    Q_OBJECT
public:
    FractMainWindow();

signals:
    void aboutToClose();

protected:
    virtual bool queryClose();
};

class KFract : public KApplication
{
    Q_OBJECT
public:
    KFract( );

private:
    double center, width, height;
    FractMainWindow *w;
    KAction *mandel, *julia;
    KAccel *accel;
    DrawView *v;
    void setupMenuBar();
    void setupToolBar();
private slots:
    void keyBindings();
    void loadParam();
    void saveParam();
    void savePic();
//    void aboutFract();

```

```

void helpFract();
void mandelFract();
void juliaFract();
void iterFract();
void zoomFract();
void my_quit();
void willClose();
void getIter( int new_iter );
void getBailout ( double new_bailout );
void getCenterX( double new_center_x );
void getCenterY( double new_center_y );
void getWidth( double new_width );
void getExtraX( double extra_x );
void getExtraY( double extra_y );
void getZoomIn( double zoom_in );
void getZoomOut( double zoom_out );
void getColorScheme( DrawView::ColorScheme new_scheme );
void changedToMandel();
void changedToJulia();
void doZoomIn();
void doZoomOut();
void goHome();
void notImplemented();
void colorSchemeFract();
};

#endif // KFRACT_H

```

PFRACT.CPP

```

#include <iostream.h>
#include <qkeycode.h>

#include "kfract.moc"
#include "version.h"

#include <klocale.h>
#include <kiconloader.h>
#include <kmessagebox.h>
#include <kmenubar.h>

#include <kaction.h>
#include <kstdaction.h>

FractMainWindow::FractMainWindow() :
    KMainWindow( 0, "kfract" )
{
}

bool FractMainWindow::queryClose()
{
    emit aboutToClose();
}

```

```

    return true;
}

KFract::KFract( )
{
    center = width = height = 0.0;
    w = new FractMainWindow();
    CHECK_PTR( w );
    setMainWidget( w );
    connect( w, SIGNAL( aboutToClose() ),
            this, SLOT( willClose() ) );

    v = new DrawView( w );
    CHECK_PTR( v );

    KAction *action;

    // keyboard accelerators
    accel = new KAccel(w);
    accel->insertItem(i18n("Mandelbrot Set"), "options_mandel",
Qt::CTRL+Qt::Key_M);
    accel->insertItem(i18n("Julia Set"), "options_julia",
Qt::CTRL+Qt::Key_J);
    accel->insertItem(i18n("Save Pic"), "file_save_pic",
Qt::CTRL+Qt::Key_P);
    accel->readSettings();

    // file menu
    action = KStdAction::open(this, SLOT(loadParam()), w-
>actionCollection());
    action->setText(i18n("&Open params..."));
    action = KStdAction::saveAs(this, SLOT(saveParam()), w-
>actionCollection());
    action->setText(i18n("&Save params..."));
    action = new KAction(i18n("Save &pic..."), "filesave",
                        accel->currentKey("file_save_pic"), this,
                        SLOT(savePic()), w->actionCollection(),
                        "file_save_pic");
    action = KStdAction::quit(this, SLOT(my_quit()), w-
>actionCollection());

    // view menu
    action = KStdAction::zoomIn(this, SLOT(doZoomIn()), w-
>actionCollection());
    action = KStdAction::zoomOut(this, SLOT(doZoomOut()), w-
>actionCollection());
    action = KStdAction::zoom(this, SLOT(zoomFract()), w-
>actionCollection());

    // options menu
    mandel = new KAction(i18n("&Mandelbrot..."), "kfract",
                        accel->currentKey("options_mandel"), this,
                        SLOT(mandelFract()), w->actionCollection(),
                        "options_mandel");

```

```

julia = new KAction(i18n("&Julia..."), "kfract",
    accel->currentKey("options_julia"), this,
    SLOT(juliaFract()), w->actionCollection(),
    "options_julia");
action = new KAction(i18n("&Iterations..."), 0, this,
    SLOT(iterFract()), w->actionCollection(),
    "options_iter");
// action = new KAction(i18n("Color &scheme..."), 0, this,
//     SLOT(colorSchemeFract()), w->actionCollection(),
//     "options_scheme");
action = KStdAction::keyBindings(this, SLOT(keyBindings()), w-
>actionCollection());

w->createGUI("kfract.rc");

w->setCentralWidget( v );
w->show();
w->resize( KFRACT_INITIAL_SIZE_X + KFRACT_SIZE_DIFF_X,
    KFRACT_INITIAL_SIZE_Y + KFRACT_SIZE_DIFF_Y );
};

void KFract::keyBindings()
{
    if (KKeyDialog::configureKeys(accel))
    {
        mandel->setAccel(accel->currentKey("options_mandel"));
        julia->setAccel(accel->currentKey("options_julia"));
    }
}

void KFract::mandelFract()
{
    v->setNewType( DrawView::FMandel );
    MandelDlg dlg( v->getActualIter(),
        v->getDefaultIter(),
        v->getActualBailout(),
        v->getDefaultBailout(),
        v->getActualCenterX(),
        v->getDefaultCenterX(),
        v->getActualCenterY(),
        v->getDefaultCenterY(),
        v->getActualWidth(),
        v->getDefaultWidth(),
        w );
    connect( &dlg, SIGNAL( iterChanged( int ) ),
        this, SLOT( getIter( int ) ) );
    connect( &dlg, SIGNAL( bailoutChanged( double ) ),
        this, SLOT( getBailout( double ) ) );
    connect( &dlg, SIGNAL( centerXChanged( double ) ),
        this, SLOT( getCenterX( double ) ) );
    connect( &dlg, SIGNAL( centerYChanged( double ) ),
        this, SLOT( getCenterY( double ) ) );
}

```

```

connect( &dlg, SIGNAL( widthChanged( double ) ),
        this, SLOT( getWidth( double ) ) );
connect( &dlg, SIGNAL( changedToMandel() ),
        this, SLOT( changedToMandel() ) );
dlg.exec();
v->setTypeUndo();
}

void KFract::juliaFract()
{
v->setNewType( DrawView::FJulia );
JuliaDlg dlg( v->getActualIter(),
              v->getDefaultIter(),
              v->getActualBailout(),
              v->getDefaultBailout(),
              v->getActualCenterX(),
              v->getDefaultCenterX(),
              v->getActualCenterY(),
              v->getDefaultCenterY(),
              v->getActualWidth(),
              v->getDefaultWidth(),
              v->getActualExtraX(),
              v->getDefaultExtraX(),
              v->getActualExtraY(),
              v->getDefaultExtraY(),
              w );
connect( &dlg, SIGNAL( iterChanged( int ) ),
        this, SLOT( getIter( int ) ) );
connect( &dlg, SIGNAL( bailoutChanged( double ) ),
        this, SLOT( getBailout( double ) ) );
connect( &dlg, SIGNAL( centerXChanged( double ) ),
        this, SLOT( getCenterX( double ) ) );
connect( &dlg, SIGNAL( centerYChanged( double ) ),
        this, SLOT( getCenterY( double ) ) );
connect( &dlg, SIGNAL( widthChanged( double ) ),
        this, SLOT( getWidth( double ) ) );
connect( &dlg, SIGNAL( extraXChanged( double ) ),
        this, SLOT( getExtraX( double ) ) );
connect( &dlg, SIGNAL( extraYChanged( double ) ),
        this, SLOT( getExtraY( double ) ) );
connect( &dlg, SIGNAL( changedToJulia() ),
        this, SLOT( changedToJulia() ) );
dlg.exec();
v->setTypeUndo();
}

void KFract::iterFract()
{
IterDlg dlg( v->getActualIter(),
             v->getDefaultIter(), w );
connect( &dlg, SIGNAL( iterChanged( int ) ),
        this, SLOT( getIter( int ) ) );
}

```

```

    dlg.exec();
}

void KFract::zoomFract()
{
    ZoomDlg dlg( v->getActualZoomInFactor(),
                v->getDefaultZoomInFactor(),
                v->getActualZoomOutFactor(),
                v->getDefaultZoomOutFactor(), w );
    connect( &dlg, SIGNAL( zoomInChanged( double ) ),
            this, SLOT( getZoomIn( double ) ) );
    connect( &dlg, SIGNAL( zoomOutChanged( double ) ),
            this, SLOT( getZoomOut( double ) ) );
    dlg.exec();
}

void KFract::getZoomIn( double zoom_in )
{
    v->setNewZoomInFactor( zoom_in );
}

void KFract::getZoomOut( double zoom_out )
{
    v->setNewZoomOutFactor( zoom_out );
}

void KFract::colorSchemeFract()
{
    // doesn't exist nay more
}

void KFract::getColorScheme( DrawView::ColorScheme scheme )
{
    v->setNewColorScheme( scheme );
}

void KFract::notImplemented()
{
    KMessageBox::sorry( w, i18n("Oops! Not implemented yet."));
}

void KFract::helpFract()
{
    kapp->invokeHelp( "", "" );
}

void KFract::getIter( int new_iter )

```

```
{
v->setNewIter( new_iter );
}

void KFract::getBailout( double new_bailout )
{
v->setNewBailout( new_bailout );
}

void KFract::getCenterX( double new_center_x )
{
v->setNewCenterX( new_center_x );
}

void KFract::getCenterY( double new_center_y )
{
v->setNewCenterY( new_center_y );
}

void KFract::getWidth( double new_width )
{
v->setNewWidth( new_width );
}

void KFract::getExtraX( double new_extra_x )
{
v->setNewExtraX( new_extra_x );
}

void KFract::getExtraY( double new_extra_y )
{
v->setNewExtraY( new_extra_y );
}

void KFract::my_quit()
{
v->stop();
kapp->quit();
}

void KFract::willClose()
{
v->prepareClose();
}

void KFract::changedToMandel()
```

```
{
v->changeToMandel();
}

void KFract::changedToJulia()
{
v->changeToJulia();
}

void KFract::savePic()
{
v->savePic();
}

void KFract::saveParam()
{
v->saveParam();
}

void KFract::loadParam()
{
v->loadParam();
}

void KFract::doZoomIn()
{
v->doZoomIn();
}

void KFract::doZoomOut()
{
v->doZoomOut();
}

void KFract::goHome()
{
v->goDefaults();
}
```

DRAWVIEW.H

```

#ifndef DRAWVIEW_H
# define DRAWVIEW_H

#include <qwidget.h>
#include <qpainter.h>
#include <qvector.h>

#include "mandel.h"
#include "julia.h"
#include "clusternode.h"

#define KFRACT_INITIAL_SIZE_X 400
#define KFRACT_INITIAL_SIZE_Y 300
#define KFRACT_SIZE_DIFF_TOP 0
#define KFRACT_SIZE_BORDER 2
#define KFRACT_SIZE_DIFF_X KFRACT_SIZE_BORDER
#define KFRACT_SIZE_DIFF_Y KFRACT_SIZE_DIFF_TOP+KFRACT_SIZE_BORDER

class DrawView : public QWidget
{
public:
    enum ColorScheme { Rgb, Hsv };
    enum FractRunning { FUnknown, FMandel, FJulia };
    enum MessageHeader { MFractType, MFractInit, MFractCalc, MDone };
    DrawView( QWidget *parent );
    ~DrawView();
    int getActualIter();
    int getDefaultIter();
    double getActualBailout();
    double getDefaultBailout();
    double getActualCenterX();
    double getDefaultCenterX();
    double getActualCenterY();
    double getDefaultCenterY();
    double getActualWidth();
    double getDefaultWidth();
    double getActualExtraX();
    double getDefaultExtraX();
    double getActualExtraY();
    double getDefaultExtraY();
    double getActualZoomInFactor();
    double getDefaultZoomInFactor();
    double getActualZoomOutFactor();
    double getDefaultZoomOutFactor();
    ColorScheme getActualColorScheme();
    void setNewType( FractRunning new_fract );

```

```

void setTypeUndo();
void setNewIter( int new_iter );
void setNewBailout( double new_bailout );
void setNewCenterX( double new_center_x );
void setNewCenterY( double new_center_y );
void setNewWidth( double new_width );
void setNewExtraX( double new_extra_x );
void setNewExtraY( double new_extra_y );
void setNewZoomInFactor( double new_zoom_in_factor );
void setNewZoomOutFactor( double new_zoom_out_factor );
void setNewColorScheme( ColorScheme scheme );
void changeToMandel();
void changeToJulia();
void stop();
void prepareClose();
void doZoomIn();
void doZoomOut();
void goDefaults();
void loadParam( const QString & name = QString::null );
void saveParam();
void savePic();
protected:
void paintEvent( QPaintEvent * );
void resizeEvent( QResizeEvent * );
void mousePressEvent( QMouseEvent * );
void mouseReleaseEvent( QMouseEvent * );
void mouseMoveEvent( QMouseEvent * );
private:
void drawCheck();
void draw();
void drawMap();
void magnify( QPoint center, double mag_factor );
int drawLine( int x_max, int y, int *results);
int drawPoint( int x, int y );
void reset();
void resetOnChange();
QWidget *w;
QPainter *paint;
QPainter *paint_buf;
QPixmap *drawbuffer;
Fract *fract;
Fract *test_fract;
int my_button;
QPoint point_1, point_2;
double my_width, my_height;
double center_x, center_y;
double bail_out;
int max_iter;
double fract_ratio;
double param_1, param_2;
int old_size_x, old_size_y;
double zoom_in, zoom_out;
bool draw_runs, draw_stop, draw_force;
int points_drawn;
int *my_map;

```

```

    FractRunning last_fract;
    FractRunning new_type;
    ColorScheme act_color_scheme;
    unsigned color_factor;
    void init_nodes();
    int getResult(int &lines_drawn);
    QVector<ClusterNode> nodes;

};

#endif    // DRAWVIEW_H

```

DRAWVIEW.CPP

```

#include <stdio.h>
#include <stdlib.h>

#include <qapplication.h>
#include <qimage.h>
#include <kfiledialog.h>
#include <qtextstream.h>
#include <kimageio.h>
#include <fstream>

#include "drawview.h"

#define FRACT_RATIO 0.01
#define FRACT_CENTER_X -0.5
#define FRACT_CENTER_Y -0.0
#define DEFAULT_ITER 100
#define JULIA_X 0.3
#define JULIA_Y 0.6
#define MAXCOLOR 16777215
#define PFRACT_PORT 8000

DrawView::DrawView( QWidget *parent ) :
    QWidget( parent )
{
    setBackgroundColor( black );
    w = parent;
    paint = new QPainter( this );
    CHECK_PTR( paint );
    fract = new Mandel();
    CHECK_PTR( fract );
    last_fract = FMandel;
    paint_buf = NULL;
    drawbuffer = NULL;
    my_map = NULL;
    reset();
    draw_runs = draw_stop = FALSE;
    init_nodes();
}

```

```

DrawView::~DrawView() {
    int header;
    for (unsigned int i = 0; i < nodes.count(); i++) {
        header = MDone;
        nodes[i]->send(&header, sizeof(header));
        delete nodes[i];
    }
    nodes.clear();
}

void DrawView::init_nodes() {
    nodes.resize(10);
    // nodes.insert(0, new ClusterNode("127.0.0.1", PFRACT_PORT));
    // nodes[nodes.count() - 1]->connect();
    ifstream infile("hosts");
    char hostaddr[80];

    while (infile.getline(hostaddr, 80)) {
        ClusterNode *nodeptr = new ClusterNode(hostaddr, PFRACT_PORT);
        nodeptr->connect();
        nodeptr->setTimeout(0);
        nodes.insert(nodes.count(), nodeptr);
    }
}

void DrawView::reset()
{
    my_button = NoButton;
    point_1.setX( 0 );
    point_1.setY( 0 );
    point_2.setX( 0 );
    point_2.setY( 0 );
    center_x = FRACT_CENTER_X;
    center_y = FRACT_CENTER_Y;
    my_width = KFRACT_INITIAL_SIZE_X / 100;
    my_height = KFRACT_INITIAL_SIZE_Y / 100;
    bail_out = 4.0;
    max_iter = DEFAULT_ITER;
    fract_ratio = FRACT_RATIO;
    old_size_x = KFRACT_INITIAL_SIZE_X;
    old_size_y = KFRACT_INITIAL_SIZE_Y;
    param_1 = 0.0;
    param_2 = 0.0;
    zoom_in = 0.5;
    zoom_out = 2.0;
    draw_runs = draw_stop = FALSE;
    draw_force = TRUE;
    if ( my_map != NULL )
    {
        delete my_map;
    }
    my_map = new int[old_size_x * old_size_y * sizeof( int )];
    CHECK_PTR( my_map );
    if ( drawbuffer != NULL )
    {

```

```

    delete drawbuffer;
}
drawbuffer = new QPixmap( old_size_x, old_size_y +
KFRACT_SIZE_DIFF_TOP );
CHECK_PTR( drawbuffer );
drawbuffer->fill( black );
if ( paint_buf != NULL )
{
    delete paint_buf;
}
paint_buf = new QPainter( drawbuffer );
CHECK_PTR( paint_buf );
points_drawn = 0;
center_x = fract->defaultCenterX();
center_y = fract->defaultCenterY();
act_color_scheme = Hsv;
color_factor = MAXCOLOR / max_iter;
test_fract = NULL;
}

```

```

void DrawView::resetOnChange()
{
    my_button = NoButton;
    point_1.setX( 0 );
    point_1.setY( 0 );
    point_2.setX( 0 );
    point_2.setY( 0 );
    draw_runs = draw_stop = FALSE;
    draw_force = TRUE;
    drawbuffer->fill( black );
    points_drawn = 0;
    center_x = fract->defaultCenterX();
    center_y = fract->defaultCenterY();
    param_1 = fract->defaultExtraX();
    param_2 = fract->defaultExtraY();
    my_width = fract->defaultWidth();
    my_height = ( my_width * ( height() - KFRACT_SIZE_DIFF_TOP ) ) /
width();
    fract_ratio = my_width / width();
    bail_out = fract->defaultBailout();
}

```

```

void DrawView::changeToMandel()
{
    if ( last_fract != FMandel )
    {
        last_fract = FMandel;
        if ( fract != NULL )
        {
            delete fract;
        }
        fract = new Mandel();
    }
}

```

```

    CHECK_PTR( fract );
    resetOnChange();
    param_1 = 0.0;
    param_2 = 0.0;
}
}

void DrawView::changeToJulia()
{
    if ( last_fract != FJulia )
    {
        last_fract = FJulia;
        if ( fract != NULL )
        {
            delete fract;
        }
        fract = new Julia();
        CHECK_PTR( fract );
        resetOnChange();
        param_1 = JULIA_X;
        param_2 = JULIA_Y;
    }
}

void DrawView::doZoomIn()
{
    QPoint center;

    center.setX( old_size_x / 2 );
    center.setY( old_size_y / 2 );
    magnify( center, zoom_in );
}

void DrawView::doZoomOut()
{
    QPoint center;

    center.setX( old_size_x / 2 );
    center.setY( old_size_y / 2 );
    magnify( center, zoom_out );
}

void DrawView::goDefaults()
{
    setNewIter( DEFAULT_ITER );
    setNewBailout( fract->defaultBailout() );
}

```

```

setNewCenterX( fract->defaultCenterX() );
setNewCenterY( fract->defaultCenterY() );
setNewWidth( fract->defaultWidth() );
setNewExtraX( fract->defaultExtraX() );
setNewExtraY( fract->defaultExtraY() );
}

void DrawView::paintEvent( QPaintEvent * )
{
    if ( draw_force )
    {
        if ( !draw_runs )
        {
            draw();
        }
    }
    else
    {
        drawMap();
    }
}

void DrawView::drawCheck()
{
    draw_force = TRUE;
    if ( draw_runs )
    {
        if ( draw_stop == FALSE )
        {
            draw_stop = TRUE;
        }
    }
    else
    {
        update();
    }
}

int DrawView::drawLine(int x_max, int y, int *results) {

    int result;
    QColor c;

    for (int x = 0; x < x_max; x++) {
        result = results[x];

        if ( result == max_iter ) {
            paint_buf->setPen( black );
        }
        else {
            switch ( act_color_scheme ) {
                case Rgb:

```

```

        c.setRgb( result * color_factor );
        break;
        case Hsv:
        c.setHsv( ( result % 72 ) * 5, 255, 255 );
        break;
        default:
        c.setHsv( ( result % 72 ) * 5, 255, 255 ); // just in case
...
        break;
    }
    paint_buf->setPen( c );
}
paint_buf->drawPoint( x, y );
}
return 1;
}

int DrawView::getResult(int& lines_drawn) {
    int x_max = width();
    int *map_pointer;
    int yval;
    unsigned int node;
    while (1) {
        for (node = 0; node < nodes.count(); node++) {
            if (nodes[node]->recv(&yval, sizeof(yval)) != -1)
                break;
        }
        if (node < nodes.count())
            break;
    }
    nodes[node]->setTimeout(-1);
    map_pointer = my_map + (yval - KFRACT_SIZE_DIFF_TOP) * x_max;
    nodes[node]->recv(map_pointer, x_max * sizeof(int));
    nodes[node]->setTimeout(0);
    nodes[node]->setFree();
    drawLine(x_max, yval, map_pointer);
    points_drawn += x_max;
    lines_drawn++;
    bitBlt( this, 0, yval, drawbuffer, 0, yval, x_max, 1,
CopyROP, FALSE );
    return node;
}

void DrawView::draw() {
    int x_max, y_max, actual_ymax;
    int y, lines_drawn;
    int header;

    draw_runs = TRUE;
    draw_force = FALSE;
    points_drawn = 0;
    x_max = width();
    y_max = height();

    actual_ymax = y_max - KFRACT_SIZE_DIFF_TOP;

```

```

for (unsigned int i = 0; i < nodes.count(); i++) {
    header = MFractType;
    nodes[i]->send(&header, sizeof(header));
    nodes[i]->send(&last_fract, sizeof(last_fract));

    header = MFractInit;
    nodes[i]->send(&header, sizeof(header));
    nodes[i]->send(&center_x, sizeof(center_x));
    nodes[i]->send(&center_y, sizeof(center_y));
    nodes[i]->send(&fract_ratio, sizeof(fract_ratio));
    nodes[i]->send(&fract_ratio, sizeof(fract_ratio));
    nodes[i]->send(&x_max, sizeof(x_max));
    nodes[i]->send(&actual_ymax, sizeof(actual_ymax));
    nodes[i]->send(&max_iter, sizeof(max_iter));
    nodes[i]->send(&bail_out, sizeof(bail_out));
    nodes[i]->send(&param_1, sizeof(param_1));
    nodes[i]->send(&param_2, sizeof(param_2));
}

for ( lines_drawn = 0, y = KFRACT_SIZE_DIFF_TOP; y < y_max; y++) {
    unsigned int node;
    for (node = 0; node < nodes.count() && nodes[node]->isInUse();
node++);
    if (node == nodes.count()) {
        node = getResult(lines_drawn);
    }

    header = MFractCalc;
    nodes[node]->send(&header, sizeof(header));
    nodes[node]->send(&y, sizeof(y));
    nodes[node]->setInUse();

    if ( draw_stop ) {
        draw_runs = draw_stop = FALSE;
        update();
        return;
    }
    qApp->processEvents();
}
while (lines_drawn < y_max - KFRACT_SIZE_DIFF_TOP) {
    getResult(lines_drawn);
}
draw_runs = draw_stop = FALSE;
}

/*
void DrawView::draw()
{
    int x_max, y_max;
    int x, y;
    int *map_pointer;

    draw_runs = TRUE;

```

```

draw_force = FALSE;
points_drawn = 0;
x_max = width();
y_max = height();
fract->init( center_x, center_y, fract_ratio, fract_ratio,
            x_max, y_max - KFRACT_SIZE_DIFF_TOP, max_iter, bail_out,
            param_1, param_2 );
map_pointer = my_map;
for ( y = KFRACT_SIZE_DIFF_TOP; y < y_max; y++ )
{
    for ( x = 0; x < x_max; x++ )
    {
        if ( draw_stop )
        {
            draw_runs = draw_stop = FALSE;
            update();
            return;
        }
        *map_pointer++ = drawPoint( x, y );
        points_drawn++;
    }
    bitBlt( this, 0, y, drawbuffer, 0, y,
            x_max, 1, CopyROP, FALSE );
    QApplication->processEvents();
}
draw_runs = draw_stop = FALSE;
}
*/

```

```

void DrawView::drawMap()
{
    bitBlt( this, 0, KFRACT_SIZE_DIFF_TOP,
            drawbuffer, 0, KFRACT_SIZE_DIFF_TOP,
            width(), height() - KFRACT_SIZE_DIFF_TOP,
            CopyROP, FALSE);

}

```

```

void DrawView::resizeEvent( QResizeEvent * )
{
    my_width *= ( double ) width() / old_size_x;
    my_height = ( double ) ( my_width * ( height() - KFRACT_SIZE_DIFF_TOP
) ) /

```

```

width();
old_size_x = width();
old_size_y = height() - KFRACT_SIZE_DIFF_TOP;
delete my_map;
my_map = new int[ old_size_x * old_size_y * sizeof( int )];
CHECK_PTR( my_map );
delete paint_buf;
delete drawbuffer;

```

```

    drawbuffer = new QPixmap( old_size_x, old_size_y +
KFRACT_SIZE_DIFF_TOP );
    CHECK_PTR( drawbuffer );
    drawbuffer->fill( black );
    paint_buf = new QPainter( drawbuffer );
    CHECK_PTR( paint_buf );
    points_drawn = 0;
    drawCheck();
}

```

```

void DrawView::mousePressEvent( QMouseEvent *e )
{
    int this_button;

    this_button = e->button() & MouseButtonMask;
    if ( my_button == NoButton )
    {
        switch ( this_button )
        {
            case LeftButton:
                point_1 = point_2 = e->pos();
                my_button = my_button | LeftButton;
                break;
            case MidButton:
                magnify( e->pos(), zoom_in );
                break;
            default:
                magnify( e->pos(), zoom_out );
                break;
        }
    }
}

```

```

void DrawView::mouseReleaseEvent( QMouseEvent *e )
{
    int this_button;
    int my_left, my_right, my_top, my_bottom;

    this_button = e->button() & MouseButtonMask;

    switch ( this_button )
    {
        case LeftButton:
            if ( my_button == LeftButton )
            {
                point_2 = e->pos();
                my_left = QMIN( point_1.x(), point_2.x() );
                my_right = QMAX( point_1.x(), point_2.x() );
                my_top = QMIN( point_1.y(), point_2.y() );
                my_bottom = QMAX( point_1.y(), point_2.y() );
                if ( ( my_top != my_bottom ) && ( my_left != my_right ) )
                {
                    center_x = center_x - my_width / 2.0 +

```

```

        fract_ratio * my_left +
        ( fract_ratio * ( my_right - my_left ) ) / 2.0;
center_y = center_y - my_height / 2.0 +
        fract_ratio * ( my_top - KFRACT_SIZE_DIFF_TOP )
+
        ( fract_ratio * ( my_bottom - my_top ) ) / 2.0;
width();
my_width = ( my_width * ( my_right - my_left ) ) /
) ) /
width();
        fract_ratio = my_width / width();
        drawbuffer->fill( black );
        drawCheck();
    }
    else
    {
        bitBlt( this, my_left, my_top,
                drawbuffer, my_left, my_top, my_right - my_left +
1,
                my_bottom - my_top +
1,
                CopyROP, FALSE );
    }
}
my_button = NoButton;
break;
case MidButton:
    my_button &= ~MidButton;
    break;
case RightButton:
    my_button &= ~RightButton;
    break;
default:
    break;
}
}

void DrawView::mouseMoveEvent( QMouseEvent *e )
{
    int x1, x2, y1, y2;
    int current_x, current_y;

    current_x = e->x();
    current_y = e->y();
    if ( ( my_button & LeftButton ) == LeftButton )
    {
        x1 = QMIN( point_1.x(), point_2.x() );
        x2 = QMAX( point_1.x(), point_2.x() );
        y1 = QMIN( point_1.y(), point_2.y() );
        y2 = QMAX( point_1.y(), point_2.y() );
        bitBlt( this, x1, y1, drawbuffer, x1, y1, x2 - x1 + 1, y2 - y1 + 1,
                CopyROP, FALSE );
    }
}

```

```

    paint->setPen( white );
    paint->drawLine( point_1.x(), point_1.y(), current_x, point_1.y()
);
    paint->drawLine( point_1.x(), current_y, current_x, current_y );
    paint->drawLine( point_1.x(), point_1.y(), point_1.x(), current_y
);
    paint->drawLine( current_x, point_1.y(), current_x, current_y);
    point_2.setX( current_x );
    point_2.setY( current_y );
}
}

int DrawView::drawPoint( int x, int y )
{
    int result;
    QColor c;

    result = fract->calcPoint( x, y - KFRACT_SIZE_DIFF_TOP );
    if ( result == max_iter )
    {
        paint_buf->setPen( black );
    }
    else
    {
        switch ( act_color_scheme )
        {
            case Rgb:
                c.setRgb( result * color_factor );
                break;
            case Hsv:
                c.setHsv( ( result % 72 ) * 5, 255, 255 );
                break;
            default:
                c.setHsv( ( result % 72 ) * 5, 255, 255 ); // just in case
...
                break;
        }
        paint_buf->setPen( c );
    }
    paint_buf->drawPoint( x, y );
    return result;
}

void DrawView::magnify( QPoint center, double mag_factor )
{
    center_x += fract_ratio * center.x() - my_width / 2.0;
    center_y += fract_ratio * ( center.y() - KFRACT_SIZE_DIFF_TOP ) -
                my_height / 2.0;

    my_width *= mag_factor;
    // We do it this way in order to get the X/Y ration straight regardless
    how
    // often we zoom in or out.

```

```

    my_height = ( my_width * ( height() - KFRACT_SIZE_DIFF_TOP ) ) /
width();
    fract_ratio = my_width / width();
    drawbuffer->fill( black );
    drawCheck();
}

```

```

int DrawView::getActualIter()
{
    int rc;

    if ( last_fract == new_type )
    {
        rc = max_iter;
    }
    else
    {
        if ( new_type == FUnknown )
        {
            rc = max_iter;
        }
        else
        {
            rc = DEFAULT_ITER;
        }
    }
    return rc;
}

```

```

int DrawView::getDefaultIter()
{
    return DEFAULT_ITER;
}

```

```

double DrawView::getActualBailout()
{
    double rc;

    if ( last_fract == new_type )
    {
        rc = bail_out;
    }
    else
    {
        rc = test_fract->defaultBailout();
    }
    return rc;
}

```

```
double DrawView::getDefaultBailout()
{
    return test_fract->defaultBailout();
}
```

```
double DrawView::getActualCenterX()
{
    double rc;

    if ( last_fract == new_type )
    {
        rc = center_x;
    }
    else
    {
        rc = test_fract->defaultCenterX();
    }
    return rc;
}
```

```
double DrawView::getDefaultCenterX()
{
    return test_fract->defaultCenterX();
}
```

```
double DrawView::getActualCenterY()
{
    double rc;

    if (last_fract == new_type )
    {
        rc = center_y;
    }
    else
    {
        rc = test_fract->defaultCenterY();
    }
    return rc;
}
```

```
double DrawView::getDefaultCenterY()
{
    return test_fract->defaultCenterY();
}
```

```
double DrawView::getActualWidth()
{
```

```

double rc;

if ( last_fract == new_type )
{
    rc = my_width;
}
else
{
    rc = test_fract->defaultWidth();
}
return rc;
}

double DrawView::getDefaultWidth()
{
    return test_fract->defaultWidth();
}

double DrawView::getActualExtraX()
{
    double rc;
    if ( last_fract == new_type )
    {
        rc = param_1;
    }
    else
    {
        rc = test_fract->defaultExtraX();
    }
    return rc;
}

double DrawView::getDefaultExtraX()
{
    return test_fract->defaultExtraX();
}

double DrawView::getActualExtraY()
{
    double rc;

    if ( last_fract == new_type )
    {
        rc = param_2;
    }
    else
    {
        rc = test_fract->defaultExtraY();
    }
    return rc;
}

```

```

double DrawView::getDefaultExtraY()
{
    return test_fract->defaultExtraY();
}

double DrawView::getActualZoomInFactor()
{
    return zoom_in;
}

double DrawView::getDefaultZoomInFactor()
{
    return 0.5;
}

double DrawView::getActualZoomOutFactor()
{
    return zoom_out;
}

double DrawView::getDefaultZoomOutFactor()
{
    return 2.0;
}

DrawView::ColorScheme DrawView::getActualColorScheme()
{
    return act_color_scheme;
}

void DrawView::setNewType( FractRunning new_fract )
{
    new_type = new_fract;
    if ( test_fract != NULL )
    {
        delete test_fract;
    }
    switch ( new_fract )
    {
        case FMandel:
            test_fract = new Mandel();
            break;
        case FJulia:
            test_fract = new Julia();
            break;
        default:
            test_fract = new Mandel(); // just in case ...
    }
}

```

```

        break;
    }
}

void DrawView::setTypeUndo()
{
    new_type = FUnknown;
    if ( test_fract != NULL )
    {
        delete test_fract;
        test_fract = NULL;
    }
}

void DrawView::setNewIter( int new_iter )
{
    if ( max_iter != new_iter )
    {
        max_iter = new_iter;
        color_factor = MAXCOLOR / max_iter;
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::setNewBailout( double new_bailout )
{
    if ( bail_out != new_bailout )
    {
        bail_out = new_bailout;
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::setNewCenterX( double new_center_x )
{
    if ( center_x != new_center_x )
    {
        center_x = new_center_x;
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::setNewCenterY( double new_center_y )
{

```

```

if (center_y != new_center_y )
{
    center_y = new_center_y;
    drawbuffer->fill( black );
    drawCheck();
}
}

void DrawView::setNewWidth( double new_width )
{
    if ( my_width != new_width )
    {
        my_width = new_width;
// We do it this way in order to get the X/Y ration straight regardless
// how
// often we zoom in or out.
        my_height = ( my_width * ( height() - KFRACT_SIZE_DIFF_TOP ) ) /
width();
        fract_ratio = my_width / width();
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::setNewExtraX( double new_extra_x )
{
    if ( param_1 != new_extra_x )
    {
        param_1 = new_extra_x;
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::setNewExtraY( double new_extra_y )
{
    if ( param_2 != new_extra_y )
    {
        param_2 = new_extra_y;
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::setNewZoomInFactor( double new_zoom_in_factor )
{
    zoom_in = new_zoom_in_factor;
}

```

```

void DrawView::setNewZoomOutFactor( double new_zoom_out_factor )
{
    zoom_out = new_zoom_out_factor;
}

void DrawView::setNewColorScheme( ColorScheme scheme )
{
    if ( act_color_scheme != scheme )
    {
        act_color_scheme = scheme;
        drawbuffer->fill( black );
        drawCheck();
    }
}

void DrawView::stop()
{
    draw_stop = TRUE;
}

void DrawView::prepareClose()
{
    draw_stop = true;
    delete paint;
    paint = 0;
}

void DrawView::saveParam()
{
    QString filename;

    filename = KFileDialog::getSaveFileName( QString::null, "*.fct", this
);
    if ( filename.isNull() )
    {
        return;
    }
    QFile file( filename );
    if ( file.open ( IO_WriteOnly | IO_Truncate ) )
    {
        QTextStream t( &file );
        t.precision( 50 );
        t << "KFract-0.1.1\n";
        switch( last_fract )
        {
            case FMandel:
                t << "mandelbrot\n";
                break;
            case FJulia:

```

```

        t << "julia\n";
        break;
    default:
        t << "bullshit\n";
        break;          // How dare you!
    }
    t << center_x << "\n";
    t << center_y << "\n";
    t << my_width << "\n";
    t << my_height << "\n";
    t << bail_out << "\n";
    t << max_iter << "\n";
    t << param_1 << "\n";
    t << param_2 << "\n";
    file.close();
}
}

void DrawView::savePic()
{
    QString filename;
    QPixmap *buffer;

    filename = KFileDialog::getSaveFileName( QString::null,
    QImageIO::pattern(KImageIO::Writing), this );
    if ( !filename.isNull() )
    {
        buffer = new QPixmap( old_size_x, old_size_y );
        CHECK_PTR( buffer );
        bitBlt( buffer, 0, 0,
                drawbuffer, 0, KFRACT_SIZE_DIFF_TOP,
                old_size_x, old_size_y,
                CopyROP, FALSE );
        buffer->save( filename, QImageIO::type(filename).ascii() );
        delete buffer;
    }
}

void DrawView::loadParam( const QString & name )
{
    QString filename;
    QString line;

    if ( name == NULL )
    {
        filename = KFileDialog::getOpenFileName ( QString::null, "*.fct",
    this );
        if ( filename.isNull() )
        {
            return;
        }
    }
    else

```

```

    {
        filename = name;
    }
    QFile file( filename );
    if ( file.open( IO_ReadOnly ) )
    {
        QTextStream t( &file );
        t >> line;
        if ( line == "KFract-0.1.1" )
        {
            delete fract;
            line = QString::null;
            t >> line;
            if ( line == "mandelbrot" )
            {
                last_fract = FMandel;
                fract = new Mandel();
            }
            else
            {
                if ( line == "julia" )
                {
                    last_fract = FJulia;
                    fract = new Julia();
                }
                else
                {
                    exit ( 1 );    // Oops!
                }
            }
        }
        t >> center_x;
        t >> center_y;
        t >> my_width;
        t >> my_height;
        t >> bail_out;
        t >> max_iter;
        t >> param_1;
        t >> param_2;
        file.close();
        my_height = ( my_width * height() ) / width();
        fract_ratio = my_width / width();
        drawbuffer->fill( black );
        points_drawn = 0;
        draw_force = TRUE;
        update();
    }
}
}

```

CLUSTERNODE.H

```
#ifndef CLUSTERNODE_H
#define CLUSTERNODE_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <netdb.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>

class ClusterNode {
public:
    ClusterNode(const char* addr, int port);
    ~ClusterNode();
    int isFree();
    int isInUse();
    void setInUse();
    void setFree();
    int connect();
    int send(void *data, int len);
    int recv(void *data, int len);
    int test();
    int getTimeout();
    void setTimeout(int _timeout);
    void disconnect();

private:
    // char nodeAddr[80];
    // Socket *nodeSock;
    // int port;
    int nodeSock;
    struct sockaddr_in sockName;
    int timeout;
    int inUse;
};

#endif
```

CLUSTERNODE.CPP

```

#include "clusternode.h"

#include <string.h>

ClusterNode::ClusterNode(const char* _addr, int _port)  {

    timeout = -1;
    inUse = 0;
    struct hostent *h = ::gethostbyname(_addr);
    bzero(&sockName, sizeof(sockName));
    sockName.sin_family = AF_INET;
    sockName.sin_port = htons(_port);
    sockName.sin_addr.s_addr = ((struct in_addr *)h->h_addr)->s_addr;

    nodeSock = ::socket(AF_INET, SOCK_STREAM, 0);
    if (nodeSock == -1)
        exit(-1);

    /* strcpy(nodeAddr, _addr);
    port = _port;
    inUse = 0;
    nodeSock = new Socket(nodeAddr, port);
    nodeSock->setSoTimeout(0);*/

}

void ClusterNode::setTimeout(int _timeout) {
    timeout = _timeout;
    //nodeSock->setSoTimeout(timeout);
}

int ClusterNode::getTimeout() {
    return timeout;
}

ClusterNode::~ClusterNode(){
    if (nodeSock != -1)
        ::close(nodeSock);
    //nodeSock->close();
    //delete nodeSock;
}

int ClusterNode::isFree() {
    return !inUse;
}

int ClusterNode::isInUse() {
    return inUse;
}

void ClusterNode::setInUse() {
    inUse = 1;
}

void ClusterNode::setFree() {
    inUse = 0;
}

```

```

}

int ClusterNode::connect() {
    int val = ::connect(nodeSock, (struct sockaddr *)&sockName,
sizeof(sockName));
    return val;
    //return nodeSock->connect();
}

int ClusterNode::send(void *data, int len) {
    int count = ::send(nodeSock, data, len, 0);
    if (count != len)
        exit(-1);
    return count;
// return nodeSock->send(data, len);
}

int ClusterNode::recv(void *data, int len) {
    if (timeout == -1) {
        //while (::recv(nodeSock, data, len, MSG_PEEK) < len);
        return ::recv(nodeSock ,data, len, MSG_WAITALL);
    }
    else {
        fd_set rdfd;
        struct timeval tv;
        tv.tv_sec = 0;
        tv.tv_usec = timeout;
        FD_ZERO(&rdfd);
        FD_SET(nodeSock, &rdfd);
        if(::select(nodeSock + 1, &rdfd, NULL, NULL, &tv)) {
            if(FD_ISSET(nodeSock, &rdfd))
                return ::recv(nodeSock, data , len, MSG_WAITALL);
        }
    }
    return -1;
// return nodeSock->recv(data, len);
}

int ClusterNode::test() {
    //return nodeSock->test();
    return 1;
}

void ClusterNode::disconnect() {
    ::close(nodeSock);
    nodeSock = -1;
// nodeSock->close();
}

```

MAIN.CPP

```

#include "kfract.h"
#include "version.h"
#include <cmdlineargs.h>
#include <klocale.h>

```

```
#include <kaboutdata.h>
#include <kimageio.h>

static const char *description =
    I18N_NOOP("A Parallelized fractal generator.");

int main( int argc, char **argv )
{
    KAboutData aboutData( "kfract", I18N_NOOP("Parallel Fractal
Generator"),
        PFRACT_VERSION, description, KAboutData::License_GPL,
        "(c) 2001-2002\nSrivas N. Chennu\nVishwas N.");

    aboutData.addAuthor("Srivas N. Chennu",0, "srivasnchennu@yahoo.com");
    aboutData.addAuthor("Vishwas N.",0, "vishwas_n@hotmail.com");

    KCmdLineArgs::init( argc, argv, &aboutData );

    KFract kfract;

    KImageIO::registerFormats();

    return kfract.exec();
}
```

5.2.2. THE PFRACT SERVER

FRACT.H

```

#ifndef FRACT_H
# define FRACT_H

class Fract
{
public:
    Fract();
    virtual void init( double center_x, double center_y,
                      double dx, double dy,
                      int x_max, int y_max,
                      int iter_max, double bail_out,
                      double param_1 = 0.0, double param_2 = 0.0 );
    virtual int calcPoint( int x, int y );
    virtual double defaultCenterX();
    virtual double defaultCenterY();
    virtual double defaultWidth();
    virtual double defaultBailout();
    virtual double defaultExtraX();
    virtual double defaultExtraY();
protected:
    double x_values[2000];
    double y_values[2000];
    int max_x, max_y;
    int max_iter;
    double bail;
private:
};

#endif // FRACT_H

```

FRACT.CPP

```

#include <kapp.h>
#include <stdio.h>

#include "fract.h"
#include <klocale.h>
#include <kdebug.h>

```

```

Fract::Fract()
{
}

void Fract::init( double center_x, double center_y,
                 double dx, double dy,
                 int x_max, int y_max,
                 int iter_max, double bail_out,
                 double param_1, double param_2 )
{
    int i;
    double nonsense;

    nonsense = param_1;
    nonsense += param_2;

    max_x = x_max + 1;
    max_y = y_max + 1;
    max_iter = iter_max;
    bail = bail_out;

    for ( i = 0; i < max_x; i++ )
    {
        x_values[i] = center_x - dx * x_max / 2.0 + dx * i;
    }
    for ( i = 0; i < y_max; i++ )
    {
        y_values[i] = center_y - dy * y_max / 2.0 + dy * i;
    }
}

int Fract::calcPoint( int x, int y )
{
    int a, b;
    a = b = 0;
    a += x;
    b += y;

    kdDebug() << "Fract::calcPoint() is an abstract method. "
               "You MUST re-write it!" << endl;

    return 0;
}

double Fract::defaultCenterX()
{
    return 0.0;
}

double Fract::defaultCenterY()
{
}

```

```

return 0.0;
}

double Fract::defaultWidth()
{
return 0.0;
}

double Fract::defaultBailout()
{
return 0.0;
}

double Fract::defaultExtraX()
{
return 0.0;
}

double Fract::defaultExtraY()
{
return 0.0;
}

```

MANDEL.H

```

#ifndef MANDEL_H
# define MANDEL_H

#include "fract.h"

class Mandel : public Fract
{
public:
Mandel();
void init( double center_x, double center_y,
           double dx, double dy,
           int x_max, int y_max,
           int iter_max, double bail_out );
int calcPoint( int x, int y );
double defaultCenterX();
double defaultCenterY();
double defaultWidth();
double defaultBailout();
protected:
private:
};

```

```
#endif // MANDEL_H
```

MANDEL.CPP

```
#include <stdio.h>
```

```
#include "mandel.h"
```

```
#define DEFAULT_MANDEL_X -0.5
#define DEFAULT_MANDEL_Y 0.0
#define DEFAULT_MANDEL_WIDTH 4.0
#define DEFAULT_MANDEL_BAILOUT 4.0
```

```
Mandel::Mandel()
{
}
```

```
void Mandel::init( double center_x, double center_y,
                  double dx, double dy,
                  int x_max, int y_max,
                  int iter_max, double bail_out )
{
    Fract::init( center_x, center_y, dx, dy,
                x_max, y_max, iter_max, bail_out );
}
```

```
int Mandel::calcPoint( int x, int y )
{
    double r = x_values[x], i = y_values[y];
    double r_out = r, i_out = i;
    double tmp, r2 = r * r, i2 = i * i;
    register int iter = 0;

    while ( ( r2 + i2 < bail ) && ( iter < max_iter ) )
    {
        tmp = 2.0 * r_out * i_out + i;
        r_out = ( r2 = r_out * r_out ) - ( i2 = i_out * i_out ) + r;
        i_out = tmp;
        iter++;
    }
    return iter;
}
```

```
double Mandel::defaultCenterX()
```

```

{
return DEFAULT_MANDEL_X;
}

double Mandel::defaultCenterY()
{
return DEFAULT_MANDEL_Y;
}

double Mandel::defaultWidth()
{
return DEFAULT_MANDEL_WIDTH;
}

double Mandel::defaultBailout()
{
return DEFAULT_MANDEL_BAILOUT;
}

```

JULIA.H

```

#ifndef JULIA_H
# define JULIA_H

#include "fract.h"

class Julia : public Fract
{
public:
    Julia();
    void init( double center_x, double center_y,
              double dx, double dy,
              int x_max, int y_max,
              int iter_max, double bail_out,
              double param_x, double param_y );
    int calcPoint( int x, int y );
    double defaultCenterX();
    double defaultCenterY();
    double defaultWidth();
    double defaultBailout();
    double defaultExtraX();
    double defaultExtraY();
protected:
private:
    double julia_x, julia_y;
};

#endif // JULIA_H

```


JULIA.CPP

```

#include <stdio.h>

#include "julia.h"

#define DEFAULT_X 0.0
#define DEFAULT_Y 0.0
#define DEFAULT_WIDTH 4.0
#define DEFAULT_BAILOUT 4.0
#define DEFAULT_JULIA_X 0.3
#define DEFAULT_JULIA_Y 0.6

Julia::Julia()
{
}

void Julia::init( double center_x, double center_y,
                 double dx, double dy,
                 int x_max, int y_max,
                 int iter_max, double bail_out,
                 double param_x, double param_y )
{
    Fract::init( center_x, center_y, dx, dy,
                x_max, y_max, iter_max, bail_out );
    julia_x = param_x;
    julia_y = param_y;
}

int Julia::calcPoint( int x, int y )
{
    double r = x_values[x], i = y_values[y];
    double r_out = r, i_out = i;
    double tmp, r2 = r * r, i2 = i * i;
    register int iter = 0;

    while ( ( r2 + i2 < bail ) && ( iter < max_iter ) )
    {
        tmp = 2.0 * r_out * i_out + julia_x;
        r_out = ( r2 = r_out * r_out ) - ( i2 = i_out * i_out ) + julia_y;
        i_out = tmp;
        iter++;
    }
    return iter;
}

```

```
double Julia::defaultCenterX()
{
    return DEFAULT_X;
}
```

```
double Julia::defaultCenterY()
{
    return DEFAULT_Y;
}
```

```
double Julia::defaultWidth()
{
    return DEFAULT_WIDTH;
}
```

```
double Julia::defaultBailout()
{
    return DEFAULT_BAILOUT;
}
```

```
double Julia::defaultExtraX()
{
    return DEFAULT_JULIA_X;
}
```

```
double Julia::defaultExtraY()
{
    return DEFAULT_JULIA_Y;
}
```

PFRACSTD.H

```
#ifndef PFRACSTD_H
#define PFRACSTD_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <netdb.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/poll.h>
```

```

/**
 *@author srivas
 */

class PFractD {
public:
    PFractD(int _bindport, const char *_bindaddr = NULL);
    ~PFractD();

    int bind();
    int listen(int backlog = 0);
    int accept();
    int send(void *data, int len);
    int recv(void *data, int len);
    void setTimeout(int _timeout);
    int getTimeout();
    int close();
    const char *getClientAddr();
    int disconnect();
    int isConnected();

private:
    int svrSock, cliSock;
    int timeout;
    struct sockaddr_in sSockName, cSockName;
};

#endif

```

PFRACSTD.CPP

```

#include "pfractd.h"

PFractD::PFractD(int _bindport, const char *_bindaddr = NULL) {

    timeout = -1;
    cliSock = -1;

    struct in_addr svrAddr;

    if (_bindaddr != NULL) {
        struct hostent *h = gethostbyname(_bindaddr);
        if (h == NULL)
            svrAddr.s_addr = htonl(INADDR_ANY);
        else
            svrAddr.s_addr = ((struct in_addr *)h->h_addr)->s_addr;
    }
    else
        svrAddr.s_addr = htonl(INADDR_ANY);

    bzero(&sSockName, sizeof(sSockName));
    sSockName.sin_family = AF_INET;

```

```

sSockName.sin_port = htons(_bindport);
sSockName.sin_addr.s_addr = svrAddr.s_addr;

svrSock = socket(AF_INET, SOCK_STREAM, 0);
}

PFractD::~PFractD(){
    ::close(cliSock);
    ::close(svrSock);
}

int PFractD::bind() {
    return ::bind(svrSock, (struct sockaddr *)&sSockName,
sizeof(sSockName));
}

int PFractD::listen(int backlog = 0) {
    return ::listen(svrSock, backlog);
}

int PFractD::accept() {
    bzero(&cSockName, sizeof(cSockName));
    int cliLen = sizeof(cSockName);
    cliSock = ::accept(svrSock, (struct sockaddr *)&cSockName,
(socklen_t *)&cliLen);
    return cliSock;
}

void PFractD::setTimeout(int _timeout) {
    timeout = _timeout;
}

int PFractD::getTimeout() {
    return timeout;
}

int PFractD::send(void *data, int len) {
    int count = ::send(cliSock, data, len, 0);
    if (count != len)
        return -1;
    return count;
}

int PFractD::recv(void *data, int len) {
    if (timeout == -1) {
        // while (::recv(cliSock, data, len, MSG_PEEK) < len);
        return ::recv(cliSock, data, len, MSG_WAITALL);
    }
    else {
        struct pollfd pfd;
        pfd.fd = cliSock;
        pfd.events &= POLLIN;
        if (poll(&pfd, 1, timeout) > 0 && (pfd.revents & POLLIN) ==
POLLIN)

```

```

        return ::recv(cliSock ,data, len, 0);
    }
    return -1;
}

int PFractD::close() {
    ::close(cliSock);
    return ::close(svrSock);
}

const char *PFractD::getClientAddr() {
    return inet_ntoa(cSockName.sin_addr);
}

int PFractD::disconnect() {
    bzero(&cSockName, sizeof(cSockName));
    int ret = ::close(cliSock);
    cliSock = -1;
    return ret;
}

int PFractD::isConnected() {
    return cliSock == -1 ? 0 : 1;
}

```

MAIN.CPP

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <iostream.h>
#include <stdlib.h>
#include <signal.h>
#include "mandel.h"
#include "julia.h"
#include "pfractd.h"

#define PFRACT_PORT 8000

int* results;
Fract *fract;
PFractD *pfractd;

void sighandler(int signo) {
    cout << "Received signal " << signo << "\n";
    switch(signo) {
        case SIGPIPE: case SIGHUP:
            if (results != NULL)
                delete results;
            results = NULL;
            if (fract != NULL)
                delete fract;
            fract = NULL;
    }
}

```

```

        cout << "Disconnected from " << pfractd->getClientAddr() <<
".\n";
        pfractd->disconnect();
        return;

        case SIGQUIT: case SIGINT: case SIGTERM: case SIGSEGV:
        cout << "Pfract Server exiting... " << "\n";
        pfractd->close();
        delete pfractd;
        delete fract;
        delete results;
        exit(0);
    }
}

int main(int argc, char *argv[])
{
    enum FractRunning { FUnknown, FMandel, FJulia };
    enum MessageHeader { MUnknown, MFractType, MFractInit, MFractCalc,
MDone };

    FractRunning current_fract;
    double center_x, center_y, dx, dy, bail_out, param_1, param_2;
    int x_max, y_max, iter_max, y;
    MessageHeader msghdr, prevhdr;

    signal(SIGINT, sighandler);
    signal(SIGSEGV, sighandler);
    signal(SIGTERM, sighandler);
    signal(SIGQUIT, sighandler);
    signal(SIGHUP, sighandler);
    signal(SIGPIPE, sighandler);

    cout << "Starting the PFract Daemon...\n";
    pfractd = new PFractD(PFRACT_PORT);

    cout << "Binding to port " << PFRACT_PORT << "...\n";
    pfractd->bind();
    cout << "Switching to listen mode.\n";
    pfractd->listen();

    while (1) {
        cout << "\nWaiting for client connection...\n";
        pfractd->accept();
        cout << "Got connection from " << pfractd->getClientAddr() <<
".\n";
        msghdr = MUnknown;
        while (1) {
            if (!pfractd->isConnected())
                break;
            prevhdr = msghdr;
            pfractd->recv(&msghdr, sizeof(msghdr));
            switch(msghdr) {

```

```

case MFractType:
    pfractd->recv(&current_fract, sizeof(current_fract));
    (current_fract == FJulia) ? fract = new Julia : fract = new
Mandel;

    cout << "Fractal type set to " << current_fract << ".\n";
    break;

case MFractInit:
    if (prevhdr != MFractType) {
        cout << "Fractal type not set.\n";
        break;
    }
    pfractd->recv(&center_x, sizeof(center_x));
    pfractd->recv(&center_y, sizeof(center_y));
    pfractd->recv(&dx, sizeof(dx));
    pfractd->recv(&dy, sizeof(dy));
    pfractd->recv(&x_max, sizeof(x_max));
    pfractd->recv(&y_max, sizeof(y_max));
    pfractd->recv(&iter_max, sizeof(iter_max));
    pfractd->recv(&bail_out, sizeof(bail_out));
    pfractd->recv(&param_1, sizeof(param_1));
    pfractd->recv(&param_2, sizeof(param_2));
    cout << "Initializing fractal...\n";
    fract->init(center_x, center_y, dx, dy,
                x_max, y_max, iter_max, bail_out,
param_1, param_2);
    results = new int[x_max];
    break;

case MFractCalc:
    if (prevhdr != MFractCalc) {
        if (prevhdr == MFractInit)
            cout << "Calculating fractal color values.\n";
        else {
            cout << "Fractal not initialized.\n";
            break;
        }
    }
    pfractd->recv(&y, sizeof(y));
    fract->calcLine(x_max, y, results);
    pfractd->send(&y, sizeof(y));
    pfractd->send(results, x_max * sizeof(int));
    break;

case MDone: default:
    sighandler(SIGHUP);
    break;
    }
}
}

return EXIT_SUCCESS;
}

```

6. TESTING

The PFract system has been fully tested under the Linux Operating System. The kernel version of the Linux distribution is 2.2.4-2 (Version 7.1). The distribution was additionally running the X Windows System with the K Desktop Environment as the Window Manager. The PFract Server was running in parallel on all machines in the cluster.

Under these test conditions, the PFract system significantly improved the speed of fractal generation, as compared to the performance on a single machine. Quantitative estimates of the performance increase showed about 60 – 70% decrease in the time taken for fractal drawing. Furthermore, the performance increased linearly with the number of computing nodes in the cluster.

7. CONCLUSION

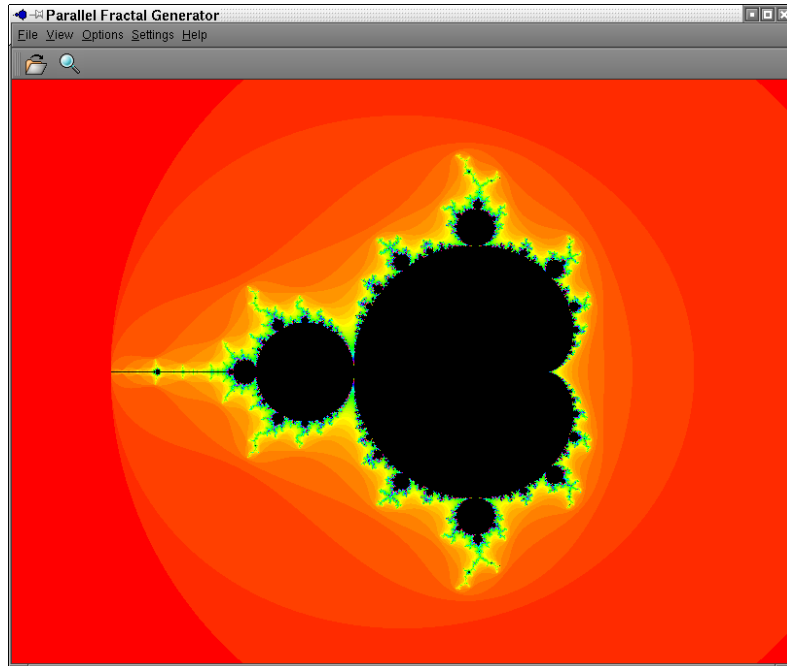
The PFract parallel computing system was successfully designed, implemented and tested. It has demonstrated the significant performance advantages parallel computing offers for solving complex computing problems.

Furthermore, the Pfract system has also displayed the potential offered by cluster computing using the Linux Operating System. The high scalability, flexibility and robustness of Linux clusters make them ideal solutions for large scale, off-the-shelf parallel computing.

THE END

APPENDIX

Below are some screenshots of the PFract system.



The PFract Client

```
Starting the PFract Server...
Binding to port 8000.
Switching to listen mode.

Waiting for client connection...
Got connection from 127.0.0.1.
Fractal type set to 1.
Initializing fractal...
Calculating fractal color values.
Fractal type set to 1.
Initializing fractal...
Calculating fractal color values.
Received signal 1
Disconnected from 127.0.0.1.

Waiting for client connection...
Got connection from 127.0.0.1.
Fractal type set to 1.
Initializing fractal...
Calculating fractal color values.
```

The PFract Server