CS/ECE 347, Spring 1996
Lab Assignment 3:
Understanding the performance of an Internet application

Assigned: Tue., Apr. 6
Due: Thu., May 2

1. Introduction

In this lab, you will create a client/server program on the Internet that computes and displays the Mandelbrodt set. The basic problem you will solve is how to partition the work between a relatively slow machine (a DECStation 3100) and a relatively fast machine (a SparcStation 5) so as to minimize running time. The solution requires an understanding of some nontrivial tradeoffs between computing power, and network and memory bandwidth. In the process, you will also have a chance to learn about the basic techniques that underly all Internet applications.

2. The Internet

The *Internet* is a world wide collection of interconnected computer networks. Any *host* computer attached to the Internet can communicate with any other Internet host. This technological marvel is made possible by a layer of software, called *TCP/IP*, that runs on each Internet host.

TCP/IP provides the fundamental capability for a process running on one host to establish a bidirectional *connection* with a process running on another host (much like a telephone connection). Once the connection is established, the processes can then reliably transfer streams of bytes back and forth over the connection (much like having a conversation over a telephone connection). When the processes have finished communicating, they can then close the connection (much like hanging up the phone).

A basic computing model for Internet processes is known as the *client/server model*. A *server* is a process that provides some resource for one or more *client* processes. Examples of resources and their servers include files provided by an AFS file server, Web pages provided by a Web server, and compute cycles provided by a compute server (which is the kind of server you will use in this lab).

In general, a server runs continuously, waiting for a request from some client to create a connection (like dialing a phone number). When a request arrives at the server, it completes the connection (like answering the phone), and then provides some service for the client (possibly using the connection). When the server has finished, the client and server close their respective ends of the connection (like hanging up the phone).

Each Internet host has a unique *name* h, where h is a character string such as *maelstrom.weh.andrew.cmu.edu*, and each server running on a host listens for connection requests on some unique (to that host) port p, where p is a positive integer. For example, most Web servers listen for connections on port 80. A tuple (h, p) uniquely identifies some server process and has the same purpose as a phone number in our phone call analogy. A client that wants to establish a connection with some server identifies that server with the tuple (h, p).

The C-language interface for TCP/IP on Andrew systems is called the Berkeley *sockets interface*. We have encapsulated the sockets interface into a simple set of routines that you will use to create connections between a client and a server.

3. The Mandelbrodt set

The Mandelbrodt set has fascinated people since it was discovered in the early 1980's by Benoit Mandelbrodt of IBM Research. The set has an infinitely complex internal structure, yet it has a simple definition and can be easily displayed on a bitmapped screen.

The Mandelbrodt set is an infinite collection of points in the complex plane. Each point $c = (x, y)$ in the complex plane corresponds to a complex number $x + yi$, where $i = \sqrt{-1}$. Given a point $c = (x, y)$, where c is in the domain $-2 < x < 1$ and $-1 < y < -1$, and given the following recurrence:

$$\begin{aligned} z_0 &= 0 \\ z_{k+1} &= z_k^2 + c \ , k = 0, 1, 2, \cdots \end{aligned} \tag{1}$$

we say that c is a member of the Mandelbrodt set iff $|z_k| < 2$ for all $k \geq 0$, where z is a complex number $(a + bi)$ and $|z| = \sqrt{a^2 + b^2}$.

For more information and some pretty pictures see the following Web pages:

`http://www.ee.ualberta.ca/~dawe/mandel.html`
`http://www.softlab.ntua.gr/mandel/mandel.html`

4. Lab basics

There are 100 total points.

You may work in a group of up to 3 people for this lab. Each group should hand in one written lab report.

Develop and debug your client/server program on any Unix machine you wish. The best approach is to run both your client and server processes on the same machine while you are debugging. However, when you are taking measurements for your lab report, you **MUST** use a DecStation 3100 (one of the eight in weh5406) for the client and a SparcStation 5 for the server.

The DecStations and SparcStations must be logged into locally and not telneted to, so you are strongly encouraged to start the Lab early to avoid contention for workstations.

Graphs should be produced by a plotting tool like *gnuplot*. They should not be hand–drawn. Also, if the numbers on the y axis differ by a factor of 10 or more, use a log scale. Axes should be clearly labeled and each graph should have a descriptive title.

When gathering your measurements, follow these guidelines:

- Use *init_etime()* and *get_etime* when measuring things on one host. Use *init_etime_real* and *get_etime_real* when measuring network performance between two processes.
- "Mortis" away any other users that are on the machine.
- Kill all other applications. This includes but is not limited to: console, clock, zwatch, emacs, netscape, messages, framemaker, background processes, etc. When you do the timing measurements you should only be running a few xterms and the client, server or test program.
- Make sure that the times for the tests are an order of magnitude greater than the timer period.
- Run all tests 10 times.

To get started, copy everything in

```
/afs/cs.cmu.edu/academic/class/15347-s96/public/labs/la3/
```

to your personal directory. Here is a description of the files. With the exception of the Makefile, you won't need to modify any of them. Curious students might enjoy studying the routines in *archsock.c* to learn how Internet connections are created.

- *Xdisp.c Xdisp.h:* X-client code.
- *archsock.c archsock.h:* Wrappers for the sockets calls that that create connections between the client and the server.
- *etime.c etime.h:* Timing routines.
- *lab3.c lab3.h:* Library of functions that will use to build your client process.
- *server.c:* The Mandelbrodt server.
- *server-sunos:* The server compiled for SunOS.
- *Makefile:* Makefile for building the client and server on both sunos and pmax.
- *sunos pmax clean-sunos clean-pmax:* Files used by the makefile.

5. Writing the client/server program

Your first task is to write the client process (*client.c*) and to get it working with the server (*server.c*) so that together they compute and display a recognizable Mandelbrodt set. We've provided the server, as well as the key functions (in *lab3.c*) that you will need to build a working client and display the set. Thus, building the client will mostly involve gluing the functions together.

The server

The server waits for a request from a client to test an arbitrary collection of points for membership in the Mandelbrodt set. Each request is an *index vector* of $M + 1$ complex numbers:

```
struct complex *buf;
```

where the first complex number contains header information:

```
buf[0].real  /* the number of points (M) to test */
buf[0].imag  /* max_iters for convergence testing */
&(buf[1])    /* beginning of M complex points to test */
```

The server evaluates (**??**) and returns a *membership vector* of M bytes, where each byte corresponds to an entry in the index vector. A zero entry in the membership vector indicates that the corresponding point is in the set. A nonzero entry signifies that the point is not in the set (the nonzeros could be used to produce a color display, but in this lab the display is black and white).

To compile the server for a Sparc, type "make server-sunos".

To run the server, type "server-sunos <port>", where port is an integer greater than or equal to 10000. For example, "server-sunos 10109" will start up the server on port 10109. The server must be running before the client is started.

It is not a good idea to put the server in the background because it periodically prints status and timing information to the standard output.

When the server is running, your client can repeatedly connect to it. However, only one client can be connected to the server at any point in time.

Terminate the server with Ctrl-C.

Writing the client

The client process (*client.c*) should accept the following four command line arguments:

- *host:* Internet name of the host that the server is running on. (e.g., maelstrom.weh.andrew.cmu.edu).
- *port:* integer port number of the server. This should be the same number used on the command line of the server. The server will only accept a port number greater than or equal to 10000.
- *N:* integer number of points in the Mandelbrodt set to compute.
- *M:* integer number of points ($0 \leq M \leq N$) to be computed by the server.

Here are the basic steps your client should perform:

1. Create an *index vector* of N (x,y) points, where x and y are doubles.
2. Send the first M points to the server (after byte reordering if the server has a different endienness than the client).
3. Perform the membership test on the remaining $N - M$ points.
4. Receive the membership vector for M points from the server.
5. Display the set on the screen (helpful for debugging).

Here are the constants, data structures, and functions that your client should use (as defined in lab3.h). DO NOT MODIFY ANY OF THESE.

```
#define MIN_REAL -2.0 /* extent of the horizontal axis */
#define MAX_REAL  1.0
#define MIN_IMAG -1.0 /* extent of the vertical axis */
#define MAX_IMAG  1.0

#define X_HEIGHT 400 /* size of the display window in pixels */
#define X_WIDTH 600

#define MAX_MES_SIZE 200000
```

```
typedef struct complex {  /* complex number */
  double real;
  double imag;
}complex;

void Compute(complex *points, int num_points,
     int max_iters, char *color);
void Disp_set(complex *points, int N, char *color);
int  Max_iters(int N);
void Connect(char *host, char *port);
void Disconnect();
void Receive(char *inbuf, int num_bytes);
void Send(char *outbuf, int num_bytes);
void Barrier();
void Reorder(double *little, double *big, int num_doubles);
```

The client should identify N points distributed evenly over the domain of the Mandelbrodt set, which is bounded by MAX_REAL, MIN_REAL, MIN_IMAG, and MAX_IMAG. In the complex plane the horizontal axis is the "real" axis and the vertical axis is the "imaginary" axis. So the domain is a rectangle whose lower left hand corner is (-2,-1) and whose upper right hand corner is (1,1).

By " distributed evenly" we mean that the domain should be chopped up into N squares, with some point chosen from each square. The middle point of each square is an obvious choice.

The client then sends M points to the server using the *Send()* function and computes the remaining $N - M$ points using the *Compute()* function.

IMPORTANT: For your reported measurements, the *max_iters* argument to the *Compute()* function should always be generated by a call to the *Max_iters(N)* function.

Next, the client synchronizes with the server using the *Barrier()* function. Finally it receives the results back from the server using *Receive()*, and displays the set using *Disp_set()*.

Byte ordering

The communication between client and server is complicated by the fact that the DecStation 3100 is a Little Endian machine and the SparcStation 5 is a Big Endian machine. For the server to work correctly the client must change the byte ordering of the array passed to the server to from Little Endian to Big Endian. This can be accomplished by calling the *Reorder()* function.

Compiling and running client

The makefile included in the directory assumes that the client is called *client.c*. To make the executables *client-sunos* or *client-pmax* type "make sunos" or "make pmax" respectively.

Note that the client and server can run on the same machine, which is handy for debugging. However, if you debug the client on a SparcStation, there is no need to call the *Reorder()* function. You might want to consider using *ifdefs* to avoid confusion:

```
#ifdef sunos
```

```
/* code for sunos */
#endif
#ifdef pmax
/* code for pmax */
#endif
```

When you are debugging your client, you can get a nice picture of the Mandelbrodt set by using values of $N = 240000$ and *max_iters = 50*.

Problem 1: Characterizing the performance of subsystems (40 pts.)

Once you have a working client, the goal of the lab is to determine, for the following values of N (total number of points),

$$N = \{2400, 9600, 12150\}$$

the values of M (number of points tested by the server) that minimize running time.

There are three subsystems that impact performance in this lab: processor, memory system, and network. In this problem you will characterize the performance of these three subsystems on the DecStation 3100 and the SparcStation 5.

Problem 1(a): Characterizing compute throughput (10 pts.)

For the DecStation 3100 and the SparcStation 5, measure and plot the throughput of the *Compute()* function, in units of "points tested per second", as a function of the following values of N:

$$N = \{2400, 9600, 12150\}$$

The *max_iters* argument to the *Compute()* function must be supplied by calling the *Max_iters(N)* function. The best approach for this problem is to instrument your client program with calls to the timing routines.

Hand in:

1. A separate graph for the DecStation and the Sparc, each on a separate page.

Problem 1(b): Characterizing memory throughput (20 pts.)

This problem is described in recitation assignment 9.

Based on your measurements, answer the following questions for both the DecStation 3100 and the SparcStation 5:

1. What is the line size? Why?
2. What is the L1 cache size? Why?
3. Is the L1 cache write through or write back? Why?
4. Extra credit: For the Sparc, is the L2 cache write through or write back? Why? (5 pts)

Hand in:

1. Four 3D graphs (read and write throughputs for the SparcStation 5 and DecStation 3100), each on a separate page.
2. On a separate page, a *brief* paragraph for each question above.

Problem 1(c): Characterizing network throughput (10 pts.)

On the DecStation 3100, measure and plot the throughput of the client's *Send()* routine, in units of *MBytes per second*, as a function of k, the number of bytes transferred, for the following values of k:

$$k = \{100, 10000, 100000\}$$

As with problem 1(a), you can simply instrument your client program to generate the performance data for the graph.

Hand in:

1. A graph, on a separate page, of the network throughput from the client to the server. Because network performance varies more than the other subsystems, you should plot the mean with 1-standard-deviation error bars above and below the mean.

Problem 2: Predicting running time (30 pts.)

Using your plots of compute, memory, and network throughputs in Problem 1, predict the running time of the program, where running time is defined as the duration from the start of the client, up to but not including the display of the set. Your predictions should be for the following values of N:

$$N = \{2400, 9600, 12150\}$$

and for each N, the following fractions of the total points (M/N) that are sent to the server:

$$M/N = \{0.0, 0.2, 0.4, 0.8, 1.0\}$$

Hint: Use the following model for the execution of the program: The client goes through some kind of setup phase where it prepares the index vectors, then the client and server synchronize while the client transfers the index vector to the server. The client and server then begin their computations at the same time. After the client and server finish their computations, each waits for the other to finish its computation (this is the purpose of the *Barrier()* routine. Once both have finished their respective computations, the server returns the membership vector to the client, which then displays the set.

Hand in:

1. A table of predicted running times, consisting of three rows (N) and five columns (M/N).
2. A description of the formula(s) that you used to predict running time.

Problem 3: Measuring the actual performance (30 pts.)

Measure the running time of your client/server program for the following values of N,

$$N = \{2400, 9600, 12150\}$$

and for each N, the following fractions of the total points (M/N) that are sent to the server:

$$M/N = \{0.0, 0.2, 0.4, 0.8, 1.0\}$$

Plot the measured running times (using a separate graph for each value of N) along with your predicted running times. On each graph, label the optimum predicted and actual fraction of points to send to the server.

Hand in:

1. Three 2D plots, one for each value of N, each with curves showing the predicted and actual running times, and the optimum fraction of points to send to the server marked for each curve.
2. If your predicted results are significantly different from the measured results, include of brief discussion of what you think happened.

6. Format of the lab report

Your report should have the following structure:

- Title sheet
- 1. Introduction (brief overview of the lab)
- 2. Problem 1
- 3. Problem 2
- 4. Problem 3
- 5. Conclusions (summary of results and any lessons you learned about the nature of client/server computations).
- Appendix (listing of client.c)