Old Dominion University
ODU Digital Commons

Computer Science Theses & Dissertations | Computer Science

Summer 2007

Investigating Real-Time Sonar Performance Predictions Using Beowulf Clustering

Charles Lane Cartledge
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds

Part of the Computer Sciences Commons

Recommended Citation

Cartledge, Charles L.. "Investigating Real-Time Sonar Performance Predictions Using Beowulf Clustering" (2007). Master of Science (MS), thesis, Computer Science, Old Dominion University, DOI: 10.25777/w2ry-5163
https://digitalcommons.odu.edu/computerscience_etds/50

This Thesis is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

INVESTIGATING REAL-TIME SONAR PERFORMANCE PREDICTIONS USING BEOWULF CLUSTERING

by

Charles Lane Cartledge
AEET June 1972, University of Alaska
BEET June 1974, Oregon Institute of Technology

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2007

Approved by:

Chester E. Grosch, Ph.D. (Director)

Alex Pothen, Ph.D. (Member)

Mohammad Zubair, Ph.D. (Member)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 1449364

Copyright 2007 by
Cartledge, Charles Lane

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1449364
Copyright 2008 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

INVESTIGATING REAL-TIME SONAR PERFORMANCE PREDICTIONS USING BEOWULF CLUSTERING

Charles Lane Cartledge
Old Dominion University, 2007
Director: Dr. Chester E. Grosch

Predicting sonar performance, critical to using any sonar to its maximum effectiveness, is computationally intensive and typically the results are based on data from the past and may not be applicable to the current water conditions. This paper discusses how Beowulf clustering techniques were investigated and applied to achieve real-time sonar performance prediction capabilities based on commercially off the shelf (COTS) hardware and software. A sonar system measures ambient noise in real-time. Based on the active sonar range scale, new ambient measurements can be available every 1 to 24 seconds. Traditional sonar performance prediction techniques operated serially and often took approximately 120 seconds of computing time per prediction. These predictions were outdated by potentially several sonar measurements. Using Beowulf clustering techniques, the same prediction now takes approximately 2 seconds. Analysis of measured data using a sonar hardware suite reveals that there is a set of sonar system parameters where a serial approach to sonar performance prediction is more efficient than Beowulf clustering. Using these parameters, a sonar engineer can make the best decision for system prediction capability based on the number of sonar beams and the expected operational range. The paper includes a discussion on the taxonomies of parallel computing, the historical developments leading to measuring the speed of light, and how those measurements enable acoustic paths to be computed in ocean environments.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Copyright 2007 by Charles Lane Cartledge. All rights reserved.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This thesis is dedicated to my wife and our son.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

I would like to acknowledge the support and encouragement of two organizations and three individuals. Old Dominion University provided much of the hardware and computer software for the early investigation into real-time sonar performance prediction capabilities. Along with these tangible assets, ODU created and maintained an environment where ideas could be explored and where efforts that did not succeed were not considered a failure, but rather a positive learning experience. At ODU, Computer Science Lecturer Mr. Jay Morris provided access to the hardware and software, acted as a sounding board and continually provided encouragement in face of seemingly overwhelming obstacles. Also at ODU, CS student, Mr. Kenneth Belkofer, "stood shoulder to shoulder in the trenches" to face and overcome problems with hardware settings and specifications, software installation, errors in documentation, to create not one, but several Beowulf clusters before and after the Beowulf laboratories were moved across campus. EDO Corporation provided access to hardware, software and engineering personnel. A crucial part to computing a sonar's performance prediction is the algorithm and source code to compute the probability of detection based on a set of assumptions and measurements. EDO provided access to the detailed and complex source code to compute that probability. The code represents a significant investment in intellectual property and EDO was willing to allow the code to be used to provide an authentic test environment. Mr. Mike Palmer, EDO Corporation Hardware Engineer, worked diligently to establish a test environment that would mimic a shipboard sonar system using hardware and software that is destined to be installed onboard a ship. This hardware suite was instrumental in defining the limits within which serial sonar performance prediction makes sense and beyond where Beowulf clustering was the only viable solution. This work would not have been possible without the aid, assistance, encouragement and support of these outstanding people, and I am deeply indebted to each of them.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

Section

1. INTRODUCTION 1
2. BACKGROUND OF THE RESEARCH 3
   - 2.1 PARALLEL COMPUTING 3
   - 2.2 TAXONOMIES OF PARALLEL COMPUTING 3
   - 2.3 BEOWULF CLUSTERING 5
   - 2.4 TYPES OF PROBLEMS WHERE PARALLEL APPROACH IS APPLICABLE 15
   - 2.5 SONAR SYSTEM 15
   - 2.6 REAL-TIME PROCESSING 20
3. JUSTIFICATION OF THE RESEARCH 21
4. PROBLEM DEFINITION 23
5. STATE OF THE ART 24
6. PROBLEM ANALYSIS 24
7. TECHNICAL SOLUTION 25
8. EVALUATION OF DEVELOPED SOLUTION 26
   - 8.1 COMPARISON OF SERIAL AND BEOWULF NUMERICAL RESULTS 26
   - 8.2 COMPARISON OF SERIAL AND BEOWULF EXECUTION TIMES 26
   - 8.3 SERIAL VS. BEOWULF "BREAK-EVEN" ANALYSIS 33
   - 8.4 USE MEASURED DATA TO PREDICT CHANGES IN CPU AND LAN SPEEDS 36
9. FUTURE WORK 38
10. CONCLUSIONS 39

REFERENCES 40

Appendices

A. DEVELOPMENT OF SNELL'S LAW OF SINES 42
B. SELECTED SONAR BEAM PATTERNS 51
C. BEOWULF (MPI) SOURCE CODE WRITTEN FOR THIS EFFORT 57
D. SOURCE CODE FOR JAVA BASED BEOWULF CLUSTER PERFORMANCE ESTIMATOR 86
E. COMPLETE RAY PROGRAM CONTROL FILE 99
F. SOUND SPEED PROFILES AND THEIR USE IN ACOUSTIC RAY TRACING 101
G. ACOUSTIC RAY TRACING 106
H. ROBUST LINEAR LEAST SQUARES 112

VITA 125

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

Table Page

I. Sonar Parameters And What Controls Them 17

II. Binary Matrix Of Sonar Detection Possibilities 20

III. Comparison Of Serial And Beowulf Sonar System Beam Processing 27

IV. Execution Time in Seconds of Various Beowulf Configurations 29

V. Robust Linear Curve Fitting for Selected Processor Combinations 32

VI. Number of Beams Above Which Beowulf Clustering Should be Used 34

VII. Representative Mirror Angular Velocities Based On Foucault's Apparatus 48

VIII. Index Of Refraction For Selected Substances 49

IX. Various Transducer Types And Associated Beam Patterns 53

X. Valid Range Of Various Values For Sound Speed Profile Equations 103

XI. Valid Range Of Various Values To Demonstrate Least Squares Curve Fitting 113

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure Page

1. Flynn's Parallel Computing Taxonomy 4

2. Shore's Type I Architecture 6

3. Shore's Type II Architecture 6

4. Shore's Type III Architecture 6

5. Shore's Type IV Architecture 6

6. Shore's Type V Architecture 6

7. Shore's Type VI Architecture 6

8. Top 500 Computer Architectures/Systems 9

9. Performance Developments of Top 500 Computer Systems 9

10. Source Code for the Beowulf Equivalent of "Hello World!" 11

11. Output from the Beowulf Equivalent of "Hello World!" 11

12. Source Code for Beowulf Process to Simulate System Loading 12

13. Outputs from Simplistic System Loading 14

14. Output from Realistic System Loading 14

15. Diagram Of Sonar Related Terms 16

16. Signal And Noise Levels Over Time 19

17. Probability-Density Functions Of Noise And Noise Plus Signal 19

18. Notional Comparison of Current and Anticipated Beowulf System Performance vs. Real-time 22

19. Beowulf Test Environment 28

20. Plot of Execution for Number Rays Based on Number of Processors 30

21. System Performance Based on Number of Processors 31

22. Number of Beams Where Serial Out Performs Beowulf 35

23. Estimated Effects of Increasing LAN from 100 Mbit to 1000 Mbit 37

24. Beowulf System Performance Improvement by Increasing the Number of Processors 38

25. Unit Circle Showing Three Standard Rays 42

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

26. Snell's Law Of Sines Diagram....43

27. Pierre De Fermat's Ray Diagram....44

28. Fizeau's Machine....45

29. Notational Diagram Of Foucault's Device To Measure The Speed Of Light....47

30. Snell's Ray Diagram With Refraction Indices....50

31. Simplified Diagram Showing How The Performance Of A Microphone Is Determined....51

32. Omni Directional Horizontal Beam Pattern....52

33. Representative Beam And Beam Pattern....54

34. Beam Pattern For A Conical Transducer....55

35. Beam Pattern For A Fan Transducer....55

36. Beam Pattern For An Omni Transducer....56

37. Beam Pattern For A Toroidal Transducer....56

38. Beowulf System Simulator Control Panel....87

39. Multi-year Sound Speed Profile From SE Of Bermuda....102

40. Handheld XBT Launcher....105

41. Cutaway Of An XBT In Its Launch Barrel....105

42. Complete XBT Measurement System....105

43. Surface Shadow Zone....107

44. Bottom Shadow Zone....108

45. Snell Ray Originating in Denser Medium....109

46. Sound Speed Profile Showing Two Sound Channels....111

47. Comparison of Standard and Robust Least Squares Curve Fitting....114

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SECTION 1. INTRODUCTION

Beowulf computer clustering is a technique used to have multiple low cost computers work in parallel to solve problems that are typically solved in a linear manner. Currently Beowulf clustering techniques are used to construct networks whose aggregate computing power rival the most expensive supercomputers. Acoustic ray tracing is at the heart of most current sonar performance prediction systems. Ray-tracing programs are used to predict the path that an acoustic wave would take from the sonar to an object (for example: a target, torpedo, fish, etc.) and then back to the sonar where the detection takes place. Once this predicted path is computed, mathematical terms are applied to compute the probability of detection of the assumed threat. These terms can be divided into three areas:

- Factors that remain relatively static (on the order of at least small numbers of tens of minutes): strength of the sonar, signal absorption, the target's acoustic reflectivity,
- Factors that change slowly (on the order of every few seconds): relative bottom topography, the direction in which the sonar is looking, the ocean sound speed profile, acoustic path, and
- Factors that change in real-time (on the order of many hundreds of times per second): ambient noise.

In this paper I will provide:

- A discussion on the various taxonomies of parallel computer systems,
- A Beowulf implementation of single board computers using the Message Passing Interface (MPI) protocol,
- A brief description of computational problems that have to be solved in a serial manner versus those that can be solved with parallel programming techniques,
- An appendix containing historical information relative to the origin of the single equation (Snell's Law of Sines) that is at the heart of all ray-tracing programs,
- A discussion on acoustic ray tracing in an aquatic environment,
- An introduction to the characteristics of acoustic beam patterns based on the selected piezoelectric

---

The journal model for this thesis is the IEEE/ACM Transactions on Networking.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

shapes,

- A high level description of sonar systems and how the probability of detection for the system is computed based on sonar beams,
- Measured Beowulf sonar performance prediction times versus serial implementation for the same data set, and
- An analysis of the Beowulf and serial measurements to identify system design "breakeven" points where it make sense to chose one approach over another.

This paper discusses how Beowulf clustering can be used to achieve real-time sonar performance predictions and provides an analysis of the trade-off in design between serial and parallel computation approaches.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SECTION 2. BACKGROUND OF THE RESEARCH

2.1 PARALLEL COMPUTING

Parallel computing is the application of multiple computing elements, possibly CPUs, working in concert to solve a problem. In the general sense, if the problem lends itself to a parallel computing approach, the more computing elements that can be applied to a problem, the faster an answer will be computed. In 1972 Flynn developed a taxonomy of parallel computing element and data combinations that is now known as Flynn's Taxonomy.

2.2 TAXONOMIES OF PARALLEL COMPUTING

Flynn's Taxonomy is based on the number of distinct instructions and the number of data elements the instructions operate on. Fig. 1 [9] provides diagrams of Flynn's taxonomy. Flynn's taxonomy is based on the idea of single or multiple numbers of instructions or data elements processed simultaneously. These possibilities are shown in the figure as a single or multiple arrow headed lines. Because there are two options for the instructions and for the data, there are four possible combinations. The characteristics of each combination are:

- Single Instruction Single Data (SISD): This is a classic von Neumann machine. The processor retrieves a single piece of data and uses one instruction to operate on that data. Modern implementations of SISD processors have pipelining embedded in the CPU, but there is only one CPU. A single CPU is an SISD architecture.
- Single Instruction Multiple Data (SIMD): A controlling element "clocks" a parallel collection of processors, each of which has a different data set presented to it. Each processor executes the same instruction on different data.
- Multiple Instruction Single Data (MISD): MISD may have many processing elements working in a serial manner, all of which are executing independent instructions. Where the output data from one processing element serves as the input to the next element, this is a macro-pipelining processor.
- Multiple Instructions Multiple Data (MIMD): Most current multiprocessor systems are in this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

classification. A MIMD system is a collection of processors that operate independently on separate data sets. The results of these independent operations are then combined into a single result. Operations across the processors are not lock-stepped as in the SIMD category, but can start and stop at different times and processors are able to communicate amongst themselves. Coordination between the processing elements can be implemented via Parallel Virtual Machine or Message Passing Interface (MPI) protocols. MPI was used for these investigations.

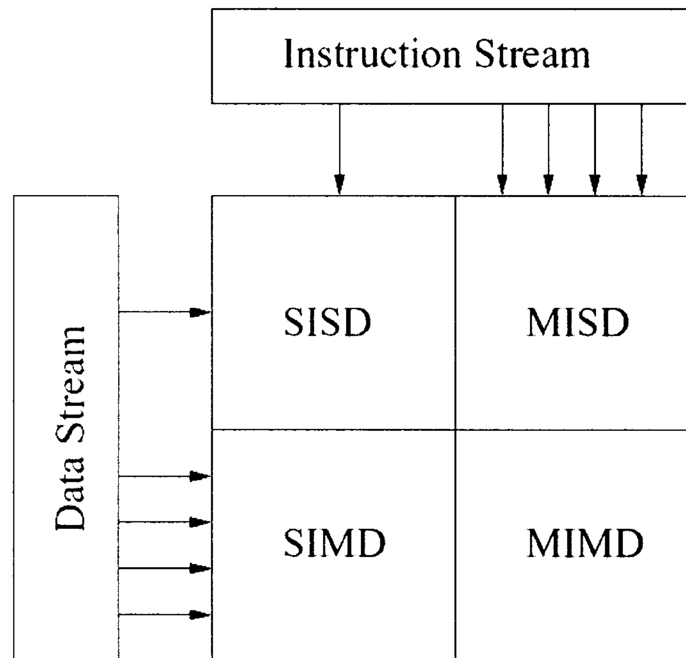


Fig. 1. Flynn's Parallel Computing Taxonomy

Shore's Taxonomy [7] of parallel computers is based on how the computer is organized from its constituent parts. Shore recognized and distinguished six different configurations and assigned each a number [7][19].

- Machine type I. Classical von Neumann architecture with a single control unit, processing unit and memory unit. Only one control unit is allowed, but it may control multiple processing units that may or may not be pipelined. The processor works on memory words and is said to be "word serial bit parallel." Examples of type I machines include the CDC 7600 and the CRAY-1. (See Fig. 2.)
- Machine type II. This type is similar to type I except that the processor works on bits-slices of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

memory rather than word-slices and is said to be "word parallel bit serial." Examples of type II machines include ICL DAP and STARAN. (See Fig. 3.)

- Machine type III. This is a combination of type I and type II machines. It has a two dimensional memory which can be read either as word or bit slices. A full implementation of a type III machine requires 2 processing units. Sanders Associates OMEN-60 series of computers were an exact implementation of type III architecture. (See Fig. 4.)
- Machine type IV. This type has a single control unit and multiple processing units. There is no communication between processing units except via the control unit. It is easy to expand a machine with this architecture by the addition of more processing units, however the communications bandwidth becomes a concern because all communications having to go through the single control unit. (See Fig. 5.)
- Machine type V. This is a machine type IV except that processing elements are able to communicate directly with their nearest neighbor. This means that a processing element can address its own memory and that of its nearest neighbor without the bandwidth limitation imposed by having to use the control unit for communications. These machines are called connected arrays. (See Fig. 6.)
- Machine type VI. Machines I to V all maintain the concept of separate data memory and processing units. Type VI machines have their processing intermeshed with the memory and are called logic in memory (LMA). (See Fig. 7.)

Shore's machines II to V are often used as subdivisions of Flynn's SIMD class. Machine I corresponds to the SISD class. Pipelined architectures are not adequately addressed by Shore's taxonomy and should be in a class by themselves.

2.3 BEOWULF CLUSTERING

2.3.1 Original Concept

The MIMD implementation currently known as Beowulf clustering is a direct result of the efforts of Donald Becker and Thomas Sterling while working at NASA in the early 1990s. As part of the Center for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

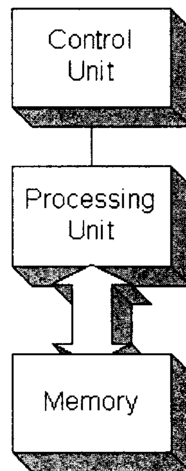


Fig. 2. Shore's Type I Architecture

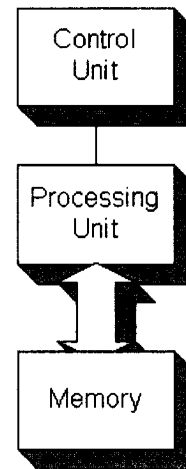


Fig. 3. Shore's Type II Architecture

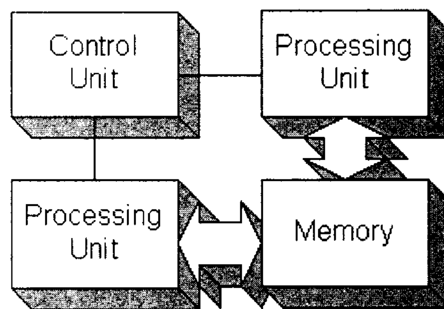


Fig. 4. Shore's Type III Architecture

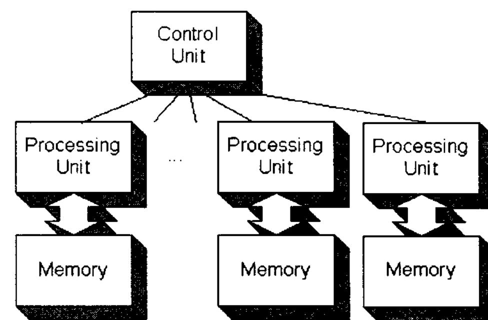


Fig. 5. Shore's Type IV Architecture

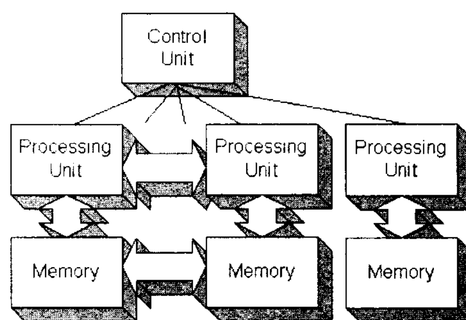


Fig. 6. Shore's Type V Architecture

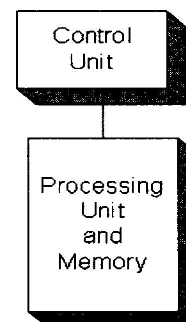


Fig. 7. Shore's Type VI Architecture

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Excellence in Space Data and Information Services (CESDIS) (Goddard Space Flight Center) portion of the Earth System Science (ESS) program, they were interested in finding a way to conduct parallel computing without using the expensive custom hardware that was the normal approach at the time.

The original Beowulf cluster was constructed with commodity off the shelf (COTS) hardware and had 16 DX4 computers communicating over a bonded Ethernet network. It was designed to help ESS personnel solve problems germane to their endeavors involving large amounts of data. The intended cluster users drove many of the initial cluster design decisions. These users were experienced parallel programming professionals that were looking for a more cost-effective solution to their computing needs than was being provided elsewhere. The users were willing to program their applications themselves in order to optimize the system's performance. This "do-it-yourself" attitude was supported by the growth of both the GNU and Linux communities.

There has been a considerable amount of folklore and rumor as to how the original COTS cluster of computers was named Beowulf; in some respects the truth is not as interesting as the lore. Dr. Thomas Sterling wrote an article [14] in which he explained the origin:

> "... In truth, I'd been struggling to come up with some cutesy acronym and failing miserably. With some small embarrassment, you can find examples of this in our early papers, which included such terms as "piles of PCs" and even "PoPC." The first term was picked up by others at least briefly. Thankfully, the second never was.
>
> Then one afternoon, Lisa, Jim Fischer's accounts manager, called me and said, "I've got to file paperwork in 15 minutes and I need the name of your project fast!" or some words to that effect. I was desperate. I looked around my office for inspiration, which had eluded me the entire previous month, and my eyes happened on my old, hardbound copy of Beowulf, which was lying on top of a pile of boxes in the corner. Honestly, I haven't a clue why it was there. As I said, I was desperate. With the phone still in my hand and Lisa waiting not all that patiently on the other end, I said, "What the hell, call it 'Beowulf.' No one will ever hear of it anyway. ..."

2.3.2 Current Concept

Fig. 8 and Fig. 9 [16] show the architectures and performance of the top 500 computers in the world as of November 2003. Fig. 8 illustrates the trend of basing more and more of the top 500 systems on Beowulf

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

clusters to achieve greater computational capacity. The constellation category is very similar to Shore's type V machine. Fig. 9 shows that while the maximum performance has remained constant for the past 3 years (as shown by the number 1 supercomputer having a performance of 35.86 TF), the spread between the performance of the top and the bottom systems has been reduced. In 1993, the least powerful of the top 500 computers to make the list was rated at 0.42 GF. Over the past 3 years, the least powerful system has increased from 100 GF to 403 GF.

2.3.3 *Implementation*

There are two basic implementations of Beowulf clustering. One implementation relies on the operating system to provide the appearance of shared memory between all processors, whereby each processor communicates with another through pseudo shared memory. The other technique, Message Passing Interface (MPI) relies on processes sending messages to other processes. The sending and receiving processes make explicit calls to message passing library routines to perform the sending and receiving of messages. Message sending was selected as the Beowulf implementation of choice for this investigation because of the minimal changes that would be required to the existing test environment. In a message passing implementation, typically there is one process that is designated as the master and all other processes are designated as slaves.

In the MPI universe of available processors, each processor listens on "a well known" port for communications from other members of the universe. A processor that initiates communication to the universe attempts to contact universe members listed in a configuration file that is available at startup. Each universe member can have different configuration files.

2.3.3.1 *Master*

The master process is responsible for tasking the other processes to complete some allocated work unit [11]. Often the master will have knowledge of how to divide the total work required into portions that slaves can work on and then consolidate their results in some manner.

2.3.3.2 *Slave*

A slave process waits to be tasked by the master process. When the slave completes some amount of work, it notifies the master that the work has been done and then waits further tasking.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

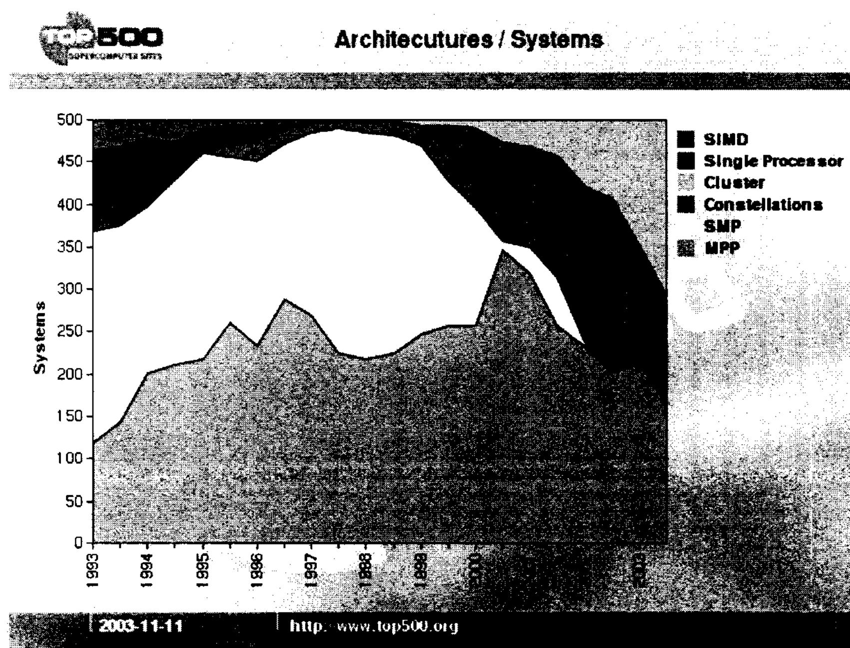


Fig. 8. Top 500 Computer Architectures/Systems

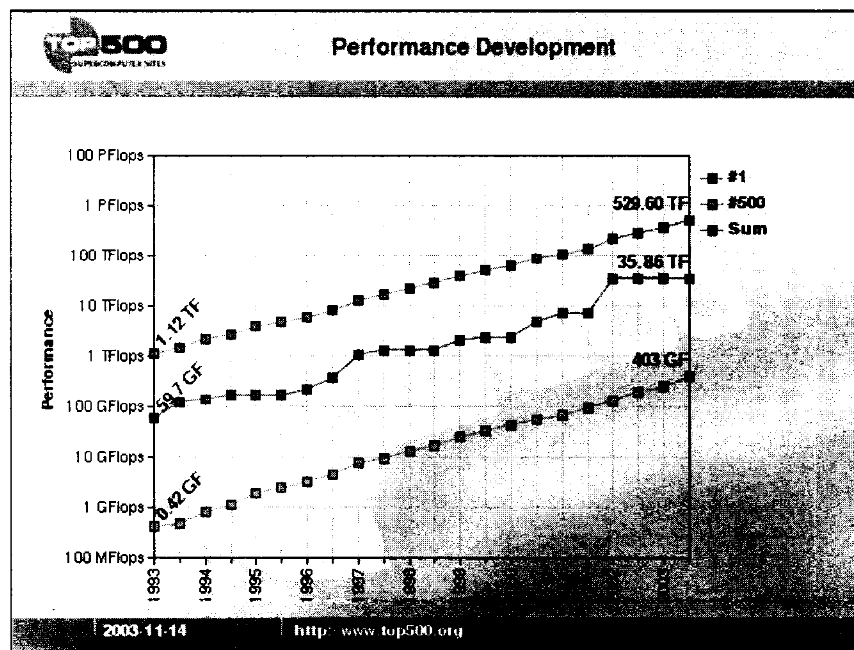


Fig. 9. Performance Developments of Top 500 Computer Systems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3.3.3 Communications between Master and Slave Processes

Communication between the master and slave processes is via asynchronous message passing. The MPI library package allows messages to be queued for delivery at a later time. Each type of process, master or slave, has to be able to deal with asynchronous communications. Fig. 10 is the source code for the Beowulf equivalent of "Hello World." Functionally, the program establishes connection to the MPI environment, determines its rank in the universe and how many processes exist there. By convention, the process whose rank is 0 is called the master. In Fig. 10, if the process is not the master, it sends a greeting message to the master. If it is the master, it prints the greeting from the slaves. In either case, the connection to the MPI universe is severed before the process terminates. Fig. 11 is the output from a small (5 process) MPI universe.

Fig. is the source code for an MPI Beowulf process that demonstrates the types of synchronization problems that can occur within a parallel execution environment, and exactly illustrates problems that occur in the final program. The master portion of the sample program distributes 15 tasks to however many slaves are available in the MPI universe. After each slave has been tasked, they are told to terminate. If more than 15 processors are available, then they will be told to terminate. The program will accept one input argument, the number of seconds that a task might take. If this argument is missing, then the task will execute as quickly as possible. In order to have some slaves take longer than others, the delay factor is doubled for any processor whose rank number sets the second bit. The output from Fig. is shown in Fig. 13 when the task timing argument is not present, while Fig. 14 shows the output when the delay is 10.

Examination of the relationship between the work unit and the time spent by the different processes on their assigned tasks reveals that processes can complete their tasks in a different sequence than the assignment of the tasks. While an individual process will complete its assigned tasks in order, there is no synchronization of sending results to the master. Sonar performance prediction relies on various computations whose running time is not known in advance and yet the correct ordering of the results is crucial to later processing. Because of this requirement, the master process must ensure that the results are in the correct order.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#include <stdio.h>

#include <string.h>     // this allows us to manipulate text strings
#include "mpi.h"        // this adds the MPI header files to the program

int main(int argc, char* argv[]) {
    int my_rank;        // process rank
    int p;              // number of processes
    int source;         // rank of sender
    int dest;           // rank of receiving process
    int tag = 0;        // tag for messages
    char message[100];  // storage for message
    MPI_Status status;  // stores status for MPI_Recv statements

    MPI_Init(&argc, &argv); // starts up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // finds out rank of each process
    MPI_Comm_size(MPI_COMM_WORLD, &p); // finds out number of processes

    if (my_rank!=0) {
        sprintf(message, "Greetings from process #%02d!", my_rank);
        // stores greeting from each process into character array
        dest = 0; // sets destination for MPI_Send to process 0
        MPI_Send(message, strlen(message)+1,
                MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        // sends the string to process 0
    } else {
        for(source = 1; source < p; source++){
          MPI_Recv(message, 100, MPI_CHAR, source,
                  tag, MPI_COMM_WORLD, &status);
          // receives greeting from each process
          printf("%s\n", message); // prints out greeting to screen
        }
    }

    MPI_Finalize(); // shuts down MPI
    return (0);
}
```

Fig. 10. Source Code for the Beowulf Equivalent of "Hello World!"

```
Greetings from process #01
Greetings from process #02
Greetings from process #03
Greetings from process #04
```

Fig. 11. Output from the Beowulf Equivalent of "Hello World!"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#include <math.h>       // this allows access to the rand() function
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     // this allows us to manipulate text strings
#include <unistd.h>     // this is allow "work" to be done
#include "mpi.h"        // this adds the MPI header files to the program

#define DIE_MESSAGE "Die spawn of Satan!"
#define FALSE        (0)
#define TRUE         (!(FALSE))

int main (int argc, char *argv[1]) {
  char hostName       [1000];  // a long hostname
  char inboundMessage [1000];  // a long input buffer
  char outboundMessage [1000]; // a long output buffer
  double delayScaleFactor;     // a way to scale processing delays
  int delayDueToWork; // time to simulate having to work
  int destination;    // rank of receiving processor
  int i;              // There will always be traces of Fortran!
  int master;         // process number of the "master"
  int messageLength;  // how long the message is
  int myRank;         // process rank
  int numberOfTasks;  // number of tasks to be done
  int processors;     // number of processors
  int source;         // rank of sending processor
  int tag;            // tag for messages
  MPI_Status status;  // stores status for MPI_Recv statements

  tag = 0;            // a way to distinguish message types
  master = 0;         // by user convention, process 0 is the master

  MPI_Init(&argc, &argv); // Connects this process to the MPI environ
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // finds my rank
  MPI_Comm_size(MPI_COMM_WORLD, &processors); // finds out number of processors
  gethostname (hostName,sizeof(hostName)-1);
  if (myRank == master){  // am I the master??
    printf ("My hostname is %s and my rank is %d. I am master.\n",
           hostName, myRank);
    numberOfTasks = 15;  // just a number
    for (i = 1, destination = 1; i <= numberOfTasks; i++) {
      sprintf (outboundMessage,"work unit #%02d",i);
      MPI_Send(outboundMessage, strlen(outboundMessage)+1,
               MPI_CHAR, destination, tag, MPI_COMM_WORLD);
      destination ++;
      if (destination == processors) destination = 1;
    }
```

Fig. 12. Source Code for Beowulf Process to Simulate System Loading

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    /* Queue the message to cause all the slaves to die. */
    strcpy (outboundMessage,DIE_MESSAGE);
    messageLength = strlen(outboundMessage) + 1;
    for (destination = 1; destination < processors; destination ++)
      MPI_Send(outboundMessage, messageLength,
               MPI_CHAR, destination, tag, MPI_COMM_WORLD);

    /* Receive messages that all the slaves have done their work. */
    for (i = 1; i <= numberOfTasks; i ++){
      MPI_Recv(inboundMessage, sizeof(inboundMessage), MPI_CHAR,
               MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
      printf ("%s\n",inboundMessage);
    }

  } else {   //  My rank is not master, so I am a working slug.
    if (argc == 2){
      delayScaleFactor = strtod(argv[1], 0);
    }else{
      delayScaleFactor = 0;
    }
    if (myRank & 2)              // to simulate different work loads
      delayScaleFactor *= 2;
    source = master;             // where we accept messages from
    destination = master;        // where we send messages to
    while (TRUE) {               // work forever
      MPI_Recv(inboundMessage, sizeof(inboundMessage),
               MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
      if (strcmp(inboundMessage,DIE_MESSAGE) == 0) {
        break;                 // we will have been told to die
      } else {
        /* Simulate doing some work here */
        delayDueToWork = rand()/(RAND_MAX + 1.0) * delayScaleFactor;
        sleep (delayDueToWork);
        sprintf (outboundMessage,
                 "Process #%02d (hostname %s) simulated working on %s for %d secs.",
                 myRank, hostName, inboundMessage, delayDueToWork);
        MPI_Send(outboundMessage, strlen(outboundMessage)+1,
                 MPI_CHAR, destination, tag, MPI_COMM_WORLD);
      }
    }
  }
  MPI_Finalize();  // break the connect to the MPI environ
  return (0);
}
```

Fig. 12. Continued.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fig. 13 shows the output of the sample code using the following command:

mpirun N ./demo

```
My hostname is system0 and my rank is 0. I am master.
Process 02 (hostname unit2) simulated working on work unit #02 for 0 secs.
Process 02 (hostname unit2) simulated working on work unit #06 for 0 secs.
Process 02 (hostname unit2) simulated working on work unit #10 for 0 secs.
Process 02 (hostname unit2) simulated working on work unit #14 for 0 secs.
Process 01 (hostname unit1) simulated working on work unit #01 for 0 secs.
Process 03 (hostname unit3) simulated working on work unit #03 for 0 secs.
Process 04 (hostname unit4) simulated working on work unit #04 for 0 secs.
Process 01 (hostname unit1) simulated working on work unit #05 for 0 secs.
Process 03 (hostname unit3) simulated working on work unit #07 for 0 secs.
Process 04 (hostname unit4) simulated working on work unit #08 for 0 secs.
Process 01 (hostname unit1) simulated working on work unit #09 for 0 secs.
Process 03 (hostname unit3) simulated working on work unit #11 for 0 secs.
Process 04 (hostname unit4) simulated working on work unit #12 for 0 secs.
Process 01 (hostname unit1) simulated working on work unit #13 for 0 secs.
Process 03 (hostname unit3) simulated working on work unit #15 for 0 secs.
```

Fig. 13. Outputs from Simplistic System Loading

Fig. 14 shows the output of the sample code using the following command:

mpirun N ./demo 10

```
My hostname is system0 and my rank is 0. I am master.
Process 01 (hostname unit1) simulated working on work unit #01 for 4 secs.
Process 04 (hostname unit4) simulated working on work unit #04 for 4 secs.
Process 01 (hostname unit1) simulated working on work unit #05 for 1 secs.
Process 04 (hostname unit4) simulated working on work unit #08 for 1 secs.
Process 02 (hostname unit2) simulated working on work unit #02 for 8 secs.
Process 03 (hostname unit3) simulated working on work unit #03 for 8 secs.
Process 01 (hostname unit1) simulated working on work unit #09 for 3 secs.
Process 04 (hostname unit4) simulated working on work unit #12 for 3 secs.
Process 02 (hostname unit2) simulated working on work unit #06 for 3 secs.
Process 03 (hostname unit3) simulated working on work unit #07 for 3 secs.
Process 01 (hostname unit1) simulated working on work unit #13 for 3 secs.
Process 02 (hostname unit2) simulated working on work unit #10 for 7 secs.
Process 03 (hostname unit3) simulated working on work unit #11 for 7 secs.
Process 02 (hostname unit2) simulated working on work unit #14 for 7 secs.
Process 03 (hostname unit3) simulated working on work unit #15 for 7 secs.
```

Fig. 14. Output from Realistic System Loading

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4 TYPES OF PROBLEMS WHERE PARALLEL APPROACH IS APPLICABLE

Parallel programming techniques are applicable to solving problems where the solution algorithm can be decomposed into steps or blocks that can be executed independently of data from other steps. The following code snippet can be implemented in a parallel manner because all data elements (a, b, and c) are independent from each other and can be represented by Shore's type IV abstraction where the computation of the value "a" could be assigned to individual processors, each of which would return their part of the total solution. The data in each step of the program execution is independent of data from any other step.

```
for (i = n; i < m; i++)
    a(i) = b(i) + c(i)
```

By way of contrast, the next code snippet is an algorithm that is not perfectly paralizable. The computation of the value of "a" depends on the value of "a" from a different iteration. While it might be technically possible to implement the algorithm in a parallel environment, it would require reworking into a different algorithmic structure.

```
for (i = n; i < m; i++)
    a(i+k) = b(i) + a(i)
```

2.5 SONAR SYSTEM

The goal of sonar is to "detect" an underwater object and provide an alert to either an operator or to another electronic system onboard the ship. Active sonar puts acoustic energy into the water and then listens for an echo. Passive sonar never puts energy into the water; it only listens for acoustic energy from the target. The amount of energy that a sonar has to receive for a detection to take place can be expressed as a sum of several individual decibels values.

Fig. 15 shows the major components of a sonar system and serves as a framework for computing the sonar's performance. TABLE I lists the various total sonar system components and indicate which parameters of the sonar system performance equation(s) are attributable to each component. In Fig. 15, a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

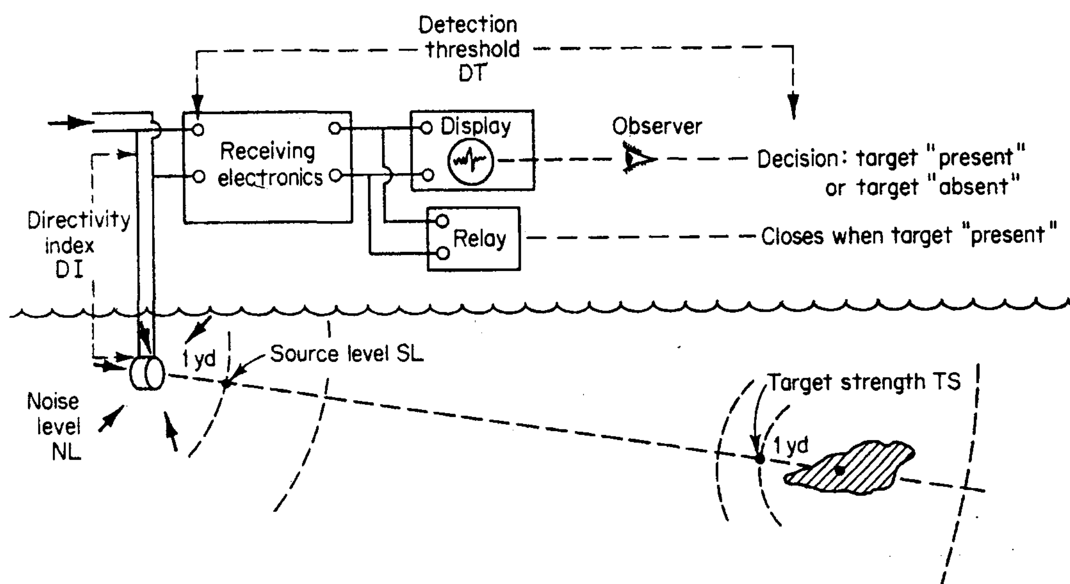


Fig. 15. Diagram Of Sonar Related Terms

single transducer shown pointed towards the target, and it is from that transducer that all measurements are made. In reality, there can be any number of transducers and they can have any shaped beam patterns. The total system performance of the sonar can be expressed by either of two equations.

They are, Eq. (1), known as the Active Sonar Equation, and Eq. (2), known as the Passive Sonar Equation [17][18]. If using either Eq. (1) or (2), DT (Detection Threshold) is less than or equal to 0 decibels, then the operator/relay will not be able to detect the signal from the target. If DT is a positive value then a signal is detected. DT is often described in terms of some positive number of decibels. For example a DT of 3dB means that the signal has to be twice (3dB = 10*LOG (x); x = 2) as strong as the background noise and a DT of 6dB would be 4 times the background level. Equations (1) and (2) hide a significant amount of complexity. The DI (Directivity Index) term includes all the gains (and losses) in the sonar system's signal processing summarized as one number.

Computing the TL (Transmission Loss) term is the focus of this investigation. TL has several components including [3]:

- Cylindrical spreading loss depending on the distance from the sonar transducer and the depth of the water,
- Range absorption based on the range from the sonar,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE I. Sonar Parameters And What Controls Them

Sonar system component	Determines these parameters in decibels
Equipment	Receiving Directivity Index: DI
	Detection Threshold: DT
	Self-Noise: NL
	Transducer/Project Source Level: SL
Water medium	Ambient-Noise Level: NL
	Reverberation Level: RL
	Transmission Loss: TL
Target	Target Source Level: SL
	Target Strength: TS

$$DT = SL - 2*TL + TS - (NL - DI) \tag{1}$$

$$DT = SL - TL - (NL - DI) \tag{2}$$

- Spherical spreading loss depending on the distance from the sonar transducer and the depth of the water,
- Viscosity losses based on the frequency of the sonar signal and an estimated speed of sound at that location,
- Depth of the sonar signal at any point in time,
- Surface bounce losses when the sonar signal bounces off the surface of the water, and
- Bottom bounce losses when the sonar signal bounces off the bottom of the ocean.

It is the modeling of the sonar's sound wave through the water medium that has to be completed in real-time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As shown in Fig. 15, the sonar operator is the final arbiter of whether or not detection is made. The operator will always make one of two decisions based on one of two conditions. The decisions are that detection is made or not made, and the conditions are the actual presence or absence of a signal from a target.

Fig. 16 (a) shows returns from three different targets. Part (b) shows the nominal background noise and part (c) is the combination of (a) and (b). M_n is the nominal noise level over time and σ_n is the one-sigma value around the mean. T_1 and T_2 represent two different levels of detection threshold. T_1 level has only one detection of possible three, but no false alarms. T_2 level detects all three and has three false alarms.

By the central limit theorem [8], the background noise M_n when measured over a relatively long time can be assumed to be Gaussian in nature with a variance σ^2 (Fig. 17).

When the input signal consists only of noise, the mean level is at M(N). When the input signal consists of noise plus signal, the mean level is at M(S+N). A threshold at level T will result in a computable probability of detection and of false alarm based on an assumption on the value of S. A measurement at the input to the sonar with amplitude along the AB line could be either noise or signal. The probability that if a signal is present that the correct decision "signal present" is made is the probability of detection p(D). The probability that if a signal is not present and the incorrect decision "signal present" decision is made, is the probability of a false alarm p(FA). A major design goal of the sonar engineer is to design the processing necessary to maximize the p(D) and minimize p(FA) by "pulling" the signal from the noise.

TABLE II summarizes the permutations of these possible inputs and decisions. As can be seen in the table, the correct decisions lie along the primary diagonal of the table and incorrect decisions along the secondary diagonal.

Eq. (3) is a redefinition of the detection threshold in terms of the relatively long-term average of noise (N) is present and the relatively long-term average of a signal (S) level in dB. Figure 16 shows three representative signal values, combining those signals with nominal background noise levels and showing the resultant signal in dB.

A sonar system will have some number of beams; the exact number is system dependent. Modeling the acoustic wave that originates at the sonar beam's transducer face is a very computationally expensive operation. An accepted simplification is to "shoot" a number of acoustic rays from the source and assume that they adequately predict the path that an acoustic wave would take. Each acoustic ray is shot at a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

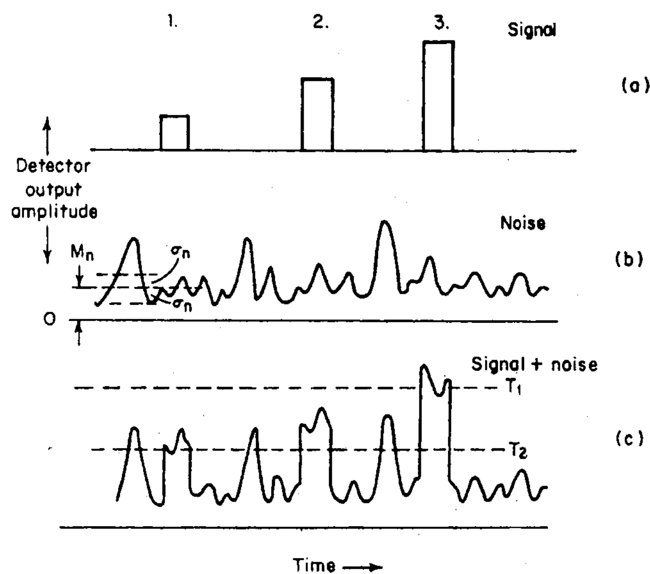


Fig. 16. Signal And Noise Levels Over Time

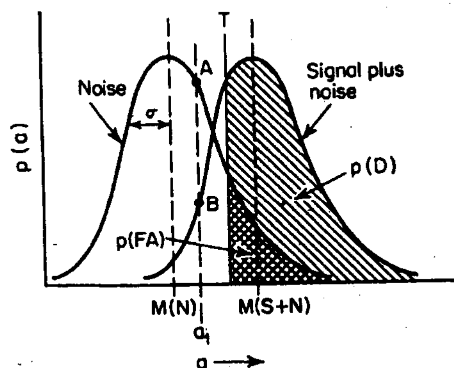


Fig. 17. Probability-Density Functions Of Noise And Noise Plus Signal

different start angle. As the ray's path is traced from the sonar to the target and back, the trace moves through sections of water that have different characteristics that affect the ray in different ways. The Ray program is used to compute the path of a ray from the sonar to the target and back. This path determines the losses in the signal relative to a constant background noise level. Levels along that path where the signal plus the noise exceeds the sonar design detection threshold will be assumed to meet the desired p(D) of the sonar.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE II. Binary Matrix Of Sonar Detection Possibilities

		Decision	
		Signal present	**Signal not present**
At the input	**Signal present**	Correct detection p(D)	Miss 1-p(D)
	Signal not present	False alarm p(FA)	Null decision 1-p(FA)

$$DT = 10 * \log(\frac{S}{N}) \quad (3)$$

2.6 REAL-TIME PROCESSING

A computer program can be considered to have operated correctly if, when given some data set, the correct answer is returned. Real-time processing adds the requirement that the correct answer be returned within some time-based criteria. For example, if a program for predicting tomorrow's weather was 100% accurate with the correct input data, but the answer took 48 hours to compute, it would not be a real-time program because it returned a correct but late answer. Real-time processing deals with issues of processing data sets at a rate that ensures that an answer is produced before the next data set arrives. The weather prediction program would be a real-time process if the answer was available before the next day's weather arrived. Real-time processing is often divided into several non-distinct divisions based on the importance of time. The areas are:

- Soft real-time: available processing time is flexible. As long as the process completes within the time constraints, it is correct and it is generally expected that the upper limit on the completion time has some amount of flexibility.
- Non-real-time: as long as the process completes it is assumed correct.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Hard real-time: available processing time is a fixed deterministic length of time; all processing has to be completed within that time, or the process fails.

Real-time processing does not relate directly to how fast a program processes data. The fundamental requirement is that the process be finished before the next data set arrives, or is needed. A process reading a deck of cards on an old style computer card deck driven computer is a real-time process if the computer can be finished with the processing before the next deck of cards is put in the card reader. A modern CPU operating at 3.1 GHz is a non real-time environment if new data arrives every millisecond and it takes two milliseconds to process.

A sonar system has sections that are hard real-time and others that are soft real-time. At the front end of the sonar, there are a number of analog to digital (A/D) converters that convert the acoustic energy from the transducers to digital data. A/D converters sample the incoming data at a fixed rate and define the hard real-time processing limit for the system. Digital Signal Processors (DSP) have to accept data from the A/D at their output rate, process the data in some manner and forward data to the later processes. DSPs often reduce the amount of data from the A/D to some lesser, but more informationally rich format. As the processing continues within the sonar from the DSP to the CRTs that display data, the real-time requirements become less and less hard. At the A/D level, real-time is defined as a 32 KHz sample rate (3.05×10^{-5} second) while at the CRT an update every 0.25-second is considered real-time. There is one process, performance prediction, which will be the focus of attention in the following paragraphs. Serial performance prediction takes one sample of data from the DSPs and processes that data for approximately 120 seconds. These serial operations are considered real-time.

SECTION 3. JUSTIFICATION OF THE RESEARCH

The normal operating environment for sonars of the type under consideration for this paper is that of searching for a target. One of the keys to best utilize the sonar is to be able to predict how the sonar will perform against the target. The performance of the sonar changes on a continuing basis due to changes in the ocean environment that are undetectable onboard the ship (for example, changes in the sound speed profile between the sonar and the target, fish and other aquatic life between the ship and the sonar, unknown differences in the bottom that affect the sonar when the acoustic wave bounces off the bottom,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

etc.). The sonar operator makes changes in the operation of the sonar (change in frequency, waveform shape, ping rate, output levels, etc.) to optimize the sonar's performance based on the performance predictions that have been computed. Ping rate is the how often the sonar "pings" per unit time. Typically the ping rate is directly related to the operational range of the sonar. The longer the range, the slower the ping rate. When the computation of the performance prediction takes longer then several sonar pings, the operator will be making decisions about how to operate the sonar based on data from several pings ago. A sonar performance prediction based on data from the last ping will enable the operator to make more timely decisions about how to employ the sonar, thereby maximizing the likelihood to detect and track the target.

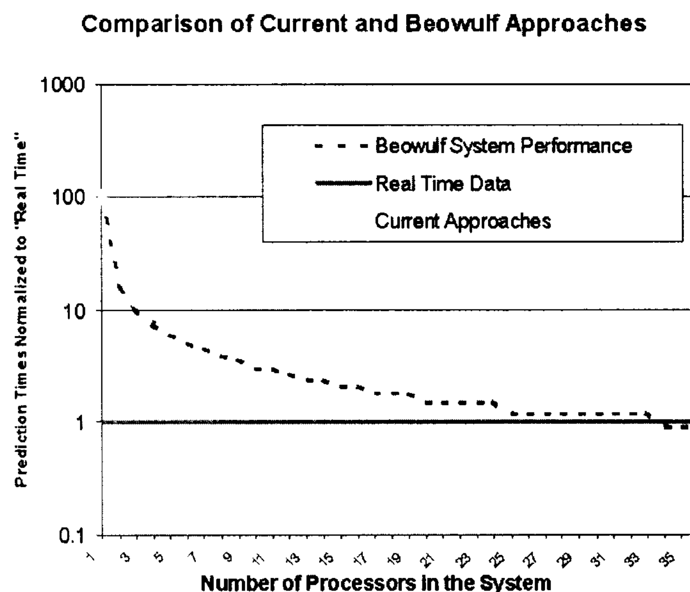


Fig. 18. Notional Comparison of Current and Anticipated Beowulf System Performance vs. Real-time

Current approaches to computing a sonar's performance, as shown in Fig. 18, are extremely slow relative to real-time ambient noise measurements. In the performance graph, all measurements are "normalized" to the real-time data rate (i.e., real-time data always arrives at 1 time unit). A single CPU approach to predicting performance takes 100 times longer than real-time, meaning that only one data sample is used for every 100 that could have been used. The predicted Beowulf approach curve shows the interaction of two quantities: the amount of computational time required to arrive at a solution and the amount of I/O time that the processors need. This estimate Eq.(4) [4] assumes no I/O conflicts and is based on:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$Time = T * (\lfloor N/S \rfloor + \lfloor N/S \rfloor \neq \lceil N/S \rceil) + N * ((I + O) / spd) \quad (4)$$

Where:

Time is the total system time to complete the problem using Beowulf techniques

T is the length of time to execute one part of the parallelized performance prediction

N is the number of parallel performance instances to execute

S is the number of slave nodes that are available

I is the size of the data sent to a Beowulf slave node

O is the size of the data sent from the slave node

spd is the measured speed of the network

The effect of floor and ceiling terms ($\lfloor\ \rfloor$ and $\lceil\ \rceil$ respectively) in Eq. (4) can most easily be seen between 21 and 25 slave processors. If N (the number of tasks to perform) is kept at 100 and:

- The number of slaves is 19 then some processors will have 6 tasks to perform, while others will have only 5 tasks,
- The number of slaves is increased to 20 then all processors will have exactly 5 tasks to perform,
- The number of slaves is increased to anywhere between 21 and 24 slaves, at least one processor will have 5 tasks, while the other processors will have only 4 tasks,
- The number of slaves is increased to 25 then all processors will have exactly 4 tasks to perform.

Within these constraints, a Beowulf system approach should demonstrate improved system performance based on the number of processors used, until real-time performance is achieved or the system becomes I/O bound. The results of a real-time Beowulf approach would be identical to a single processor approach without compromising the accuracy of the solution and would be faster by a factor 100.

SECTION 4. PROBLEM DEFINITION

The problem statement is:

> To determine if and when Beowulf clustering can be used to compute sonar performance predictions in real-time. Real-time for this context is the sonar ping rate.

An expansion of the problem statement is:

> To investigate the "parallelization" of a computationally intensive serial model of the ocean acoustic environment to support real-time data acquisition and analysis. Four areas need to be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

investigated. They are [5]:

- Decomposition: analyzing the entire performance prediction algorithm to identify those areas that can be parallelized,
- Assignment of tasks: based on the results of the decomposition analysis, possible creation of parallel routines to replace serial routines,
- Scheduling: data access, communication, and synchronization between processes, and
- Mapping of processes to processors.

SECTION 5. STATE OF THE ART

Two parts of this effort are worthy of note. One part is paralyzing the application of an acoustic ray-tracing program. The other part is the use of Beowulf clustering in a sonar related environment.

Ray [2] is a state of the art acoustic modeling program in wide use among oceanographers. The program is used to predict the paths of acoustic waves from seismic and man-made sources for short (tens of feet) and long (thousands of kilometers) ranges. Ray was developed by Woods Hole Oceanographic Institute, and is fundamental to the computation of the sonar's performance prediction.

Beowulf clustering is currently in use in some sonar related endeavors, primarily in three areas. The areas are:

- Independent robotic operations where sonar is used as the "eyes" of the robot and a cluster is used to make sense of the data,
- Image enhancements for side scan sonar systems, and
- References in the marketing literature of some defense contractors to passive and active sonar contact detection and classification.

No references were found that combined Ray and Beowulf to achieve real-time sonar performance prediction capability.

SECTION 6. PROBLEM ANALYSIS

A sonar system is composed of many different pieces of hardware and software. From a macro view, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sonar system is composed of a number of sonar beams that are processed by the sonar detection hardware. The predicted performance capability of a sonar system is a function of the predicted performance of each beam. The performance of each beam is based on the path that the acoustic wave from each beam is assumed to take. The acoustic path for each wave is assumed to be represented by the interaction of the individual rays "shot" from each beam. The path that an individual ray takes is independent of all other rays and is dependent only on its start angle and the defined environment. The mechanics of computing the performance of a sonar system conforms very nicely to Shore's Type V machine, or as a MIMD from Flynn's taxonomy.

SECTION 7. TECHNICAL SOLUTION

A sonar system will typically have more than one beam. An acoustic wave front in each beam is modeled by some number of acoustic rays. The relationship of beams and rays fits exactly into a nested loop control structure. A serial approach uses a single processor for all computations. In the limit, a Beowulf approach would use a slave processor for each ray for each beam.

The Ray program reads a configuration file containing directives about which bottom topography and which bathythermograph files to read, how many rays to shoot, how far to shoot them and other control parameters. After the individual rays are shot, the resulting wave front is used to compute p(D). **TABLE III** shows the processing that occurs in both serial and Beowulf approaches. The serial approach uses a single process to do all the work, whereas the Beowulf approach partitions the original ray initialization file into a set of files, each of which deals with a single ray. After the file is parsed, each reduced count ray file is assigned to a processor. Each of the reduced count files has a different initial start angle for its ray. Based on the start angle and the bathygraphic data, processing time may be different, so the resulting data from the individual processors have to be organized as if the entire operation was executed in a serial manner. Appendix C is a complete program listing of the code to read, parse, assign tasks and consolidate results. The source code for the final step of computing p(D) is not included because it is part of the intellectual property of EDO Corporation. EDO Corporation provided access to the final test and development environment for this effort. Fig. 19 is a representation of the test environment.

The processes used for computing the p(D) for a single beam are the same for all beams. Each beam is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

looking/listening in a different direction, but the same algorithm is used to compute p(D) for all beams. The ability of a hardware suite to compute p(D) in real-time for a single beam can be expanded to computing the p(D) for the entire sonar system by the application of additional hardware.

SECTION 8. EVALUATION OF DEVELOPED SOLUTION

Performance prediction of a sonar system is a composite of the performance prediction for each sonar beam. The time to serially compute the performance prediction is equal to the time required to compute a single beam times the number of beams in the system. The time to compute the performance prediction for the sonar system using a parallel approach is approximately the time to compute a single beam.

The proposed solution to real-time Beowulf processing was rigorously compared to the serial processing approach. The Beowulf solution was evaluated from three different perspectives: comparison of the Beowulf output to a serial output, comparison of serial and Beowulf sonar performance prediction times, determination of a "break-even" point where one approach is faster than another and use measured data to predict the effects of different CPU and LAN speeds. Each of these perspectives is addressed in the following sub-sections.

8.1 COMPARISON OF SERIAL AND BEOWULF NUMERICAL RESULTS

The output file resulting from running a serial Ray program with the test input file was compared using the emacs compare buffer command to the reconstituted Beowulf ray trace output file. There were no differences. Appendix E contains the Ray runtime file used to validate the Beowulf implementation against the serial execution.

8.2 COMPARISON OF SERIAL AND BEOWULF EXECUTION TIMES

Comparison of serial and Beowulf execution times with the same input data. Two Linux commands (time and mpirun) were used to collect all data for this analysis:

- time reports, among other things, the user time that a user supplied command (in this case: mpirun) takes to execute, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE III. Comparison Of Serial And Beowulf Sonar System Beam Processing

Serial Approach	**Beowulf Approach**	**Action performed by Master or Slave**
for each beam	for each beam	Master
Read and parse ray file	Read and parse ray file into multiple files	Master
Shoot all rays	for each ray	Master
Compute p(D)	Assign ray to slave	Master
next beam	Shoot ray	Slave
	next ray	Master
	Consolidate data (ordering of data in output file is significant)	Master
	Compute p(D)	Master
	next beam	Master

- mpirun establishes the Beowulf runtime environment on the master and slave processors and starts the processes.

During the course of the data collection, it was noticed that if there was a delay of more than about 1.5 seconds between successive executions of the time mpirun command combination, the reported time was significantly higher than if the delay had not occurred. Therefore, each combination of number of processors and representative ray counts was run at least 5 times. The first execution was to overcome the unknown initial timing delay and the remaining runs were averaged to obtain a representative result.

Data was collected for a number of different combinations of rays and available Beowulf processors. The performance of a serial task was also measured to use as a benchmark for the Beowulf approach. All rays were computed to 10 Nautical miles (Nm).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

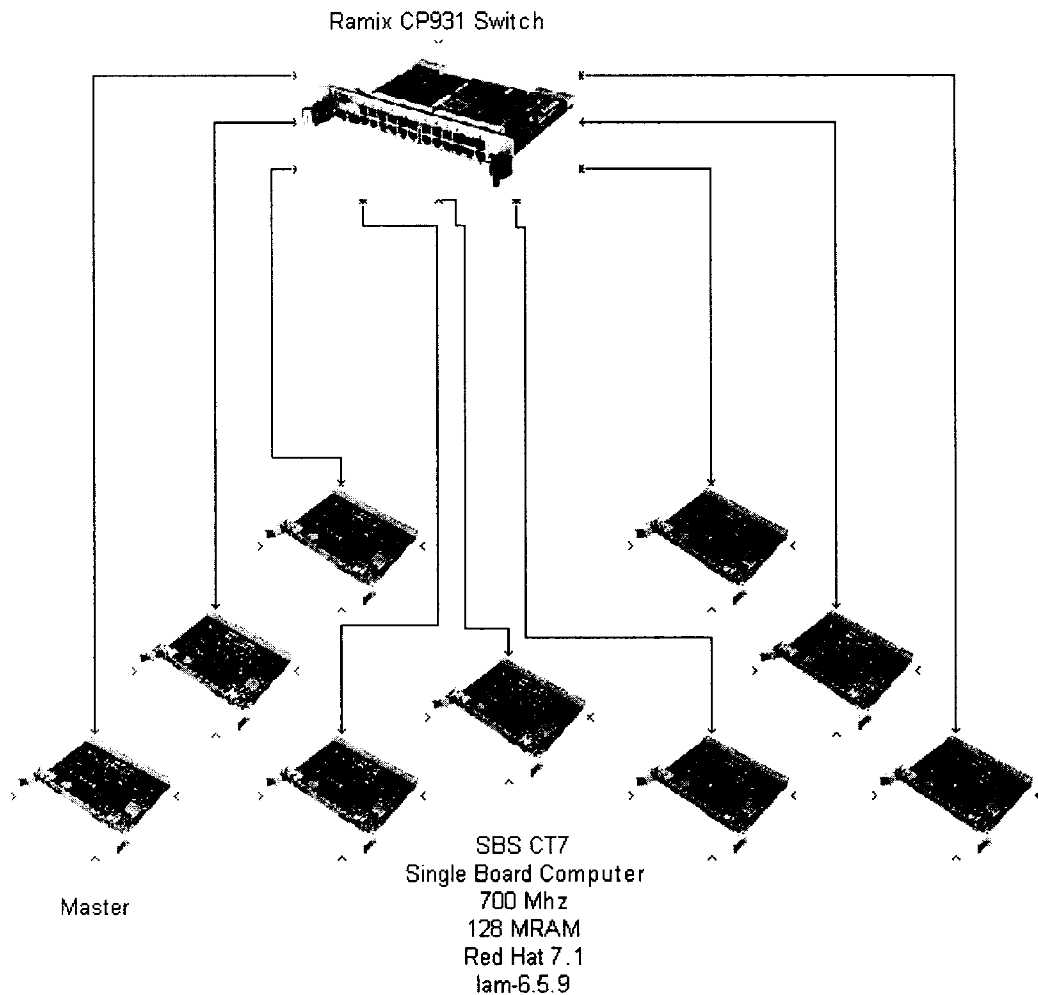


Fig. 19. Beowulf Test Environment

TABLE IV has the data that was collected and represents a single beam whose acoustic wave front is modeled with various numbers of rays.

As shown in Fig. 20, the data clusters into three rates of growth. The best performing case is the serial task, implying that the overhead of a Master and Slave implementation is greater than that of a single process. Included in the Beowulf overhead are:

- The time needed to identify which slaves are available and what OS is running on the slave,
- The communication time needed to send messages between the master and the slaves, and
- Network file system (NFS) coordination between all nodes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE IV. Execution Time in Seconds of Various Beowulf Configurations

Rays	NUMBER OF BEOWULF PROCESSORS								SERIAL TASK
	2	3	4	5	6	7	8	9	
1	0.5610	0.5690	0.5730	0.5760	0.5780	0.5860	0.5850	0.5890	0.0120
10	0.7800	0.7220	0.7110	0.7160	0.7290	0.7190	0.7240	0.7280	0.0560
20	1.0260	0.8920	0.9030	0.8700	0.8740	0.8750	1.3780	1.3280	0.1020
30	1.2830	1.0580	1.0280	1.0360	1.0330	1.0300	1.5530	1.5520	0.1510
100	3.0000	2.3280	2.2600	2.1830	2.1210	2.1860	2.6020	2.7190	0.4860
200	5.8590	4.0790	3.8950	3.9070	3.7510	3.8420	4.1820	4.2460	0.9680
1000	30.4830	17.3910	17.6340	17.2480	17.3600	17.0570	17.1760	17.6580	4.4820

The other clusters of lines in Fig. 20 are also of interest. The execution time of the 2 processors is greater than that of 3 or more slaves. This implies that there is an improvement in system performance between 2 and 3 processors, but that after 3 processors there is no measurable improvement to the limit of the test environment.

This is born out by looking at the recorded data in a different fashion. Figure 21 shows that there is a significant improvement in system performance when adding the third processor, regardless of the number of rays that are computed. From 3 to 7 processors there does not appear to be any real measurable improvement and, in fact, the system's performance decreases when the 8th and 9th processors are added.

This decrease in performance appears to be a communication channel saturation of some type, possibly either a LAN restriction or a restriction due to the NFS coordination. Because the collected data clearly shows that the system performs best with 3 to 7 processors, the rest of the analysis is restricted to this region.

The data shown in Fig. 20 appear to be along a straight line. Based on that assumption, robust linear least squares curve fitting (y = mx + b) [20] [21] was used to fit the data. A full explanation of robust linear least squares curve fitting is contained in Appendix H. The slope of the line in Table V corresponds to the m

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

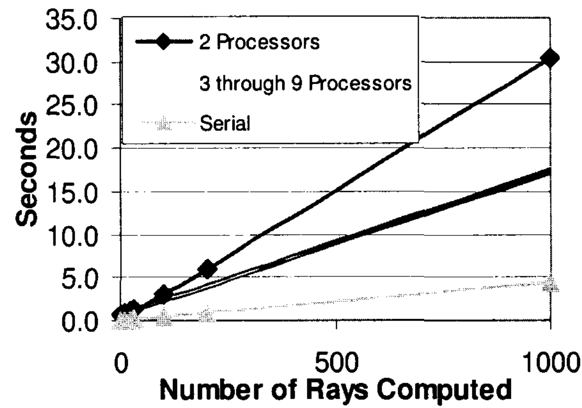


Fig. 20. Plot of Execution for Number Rays Based on Number of Processors

term in the equation y = mx + b. The average rate of growth for processors 3 through 7 for all ray combinations is 0.5300 compared to 0.0169 for the serial task. time for the serial approach is Eq. (5) and the Beowulf approach is Eq. (6).

$$tS = 0.0169 + r * 0.00045 * d \tag{5}$$

$$tB = 0.5300 + r * 0.00168 * d \tag{6}$$

Where:

tB is the Beowulf time in seconds to compute some number of rays for a given distance

tS is the serial time in seconds to compute some number of rays for a given distance

r is number of rays per beam

d is distance from the sonar to the end of the ray path (in nautical miles)

The difference between the y-axis intercepts of the two approaches (0.0169 compared to 0.5300) is likely due to the time the MPI system needs to start. As part of its startup, the processor that is executing the

TABLE V corresponds to the m term in the equation y = mx + b. The average slope for processors 3 through 7 for all ray combinations is 0.00168 compared to 0.00045 for the serial task.

Based on the above values for slope (m) and y axis intercept (b), the equations for computing the required

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

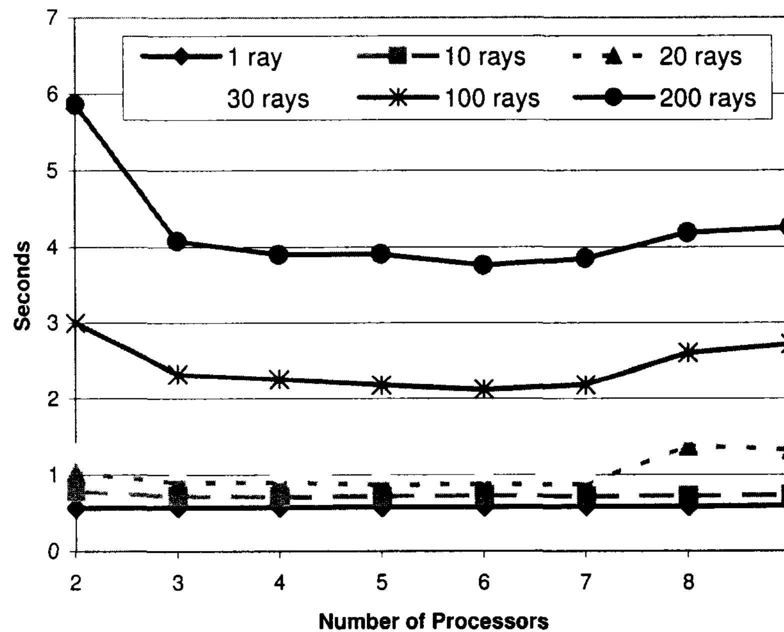


Fig. 21. System Performance Based on Number of Processors

time for the serial approach is Eq. (5) and the Beowulf approach is Eq. (6).

$$tS = 0.0169 + r * 0.00045 * d \tag{5}$$

$$tB = 0.5300 + r * 0.00168 * d \tag{6}$$

Where:

tB is the Beowulf time in seconds to compute some number of rays for a given distance

tS is the serial time in seconds to compute some number of rays for a given distance

r is number of rays per beam

d is distance from the sonar to the end of the ray path (in nautical miles)

The difference between the y-axis intercepts of the two approaches (0.0169 compared to 0.5300) is likely due to the time the MPI system needs to start. As part of its startup, the processor that is executing the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE V. Robust Linear Curve Fitting for Selected Processor Combinations

Number of Slave processors	Y axis intercept	Slope of the line
3	0.553405	0.00168376
4	0.539091	0.00170894
5	0.541148	0.00167085
6	0.476149	0.00168633
7	0.540560	0.00165107
Totals	2.650353	0.00840095
Average coefficients for a distance of 1 Nm	0.530070	0.00168019
Serial coefficients for a distance of 1 Nm	0.0168646	0.000446658

mpirun command has to query the requested number of other hosts (master and some number of slaves) to see if they are available for use. Once this startup phase is completed, the differences in the slopes of the lines come into play. The Beowulf implementation has each process run a single instance of the Ray program, and it is reasonable to assume that the execution for that single instance and the serial execution should be the same (0.00045 seconds). As the Ray program is executing, it is outputting data to a data file. In the serial execution, new ray data is appended to an already opened file. In the Beowulf implementation, each slave outputs its data to its own file. The file is then closed and the master process has to examine the data file to determine where the ray data begins and copy data from that point on to another file. This copying of data is the consolidation of data into a serial look alike form. The master opening the slave's file and searching it for data is likely to be time consuming.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8.3 SERIAL VS. BEOWULF "BREAK-EVEN" ANALYSIS

At first glance, equations (5) and (6) give a clear advantage to the serial mode of computation, but they do not reflect the practical problem that a sonar is composed of multiple beams. Therefore the serial system performance prediction time is given by Eq.(7) and the Beowulf time is given by Eq.(8).

$$timeOfSerialApproach = tS * B \tag{7}$$

$$timeOfBeowulfApproach = tB \tag{8}$$

Where:

timeOf...Approach is the total time in seconds needed to compute the performance for the sonar system

tB is from Eq.(6)

tS is from Eq.(5)

B is the number of beams in the sonar system

Examination of the relationship between equations (5) through (8) reveals that there is a breakeven point for the sonar system engineer where it makes sense to use a serial approach versus a Beowulf approach. The number of beams, the number of rays per beam and the expected operational range of the sonar determine the breakeven point in the system. Below that threshold a serial approach should be used, above that threshold a performance prediction can be made faster using a Beowulf approach TABLE VI shows the break-even point above which a Beowulf approach makes sense.

Data from TABLE VI is plotted in Fig. 22 and shows that a plane at 4 beams below which a serial approach to sonar system performance prediction is faster than a Beowulf approach. A sonar system with 4 beams implies that each beam would have covered 90 degrees to provide 360° of coverage. Most sonar systems have more than 4 beams. Specialized sonar systems with a limited number of beams, short operational ranges and the willingness to accept a small number of rays to represent the acoustic wave front could achieve real-time performance capability using a serial approach. Other systems should use a Beowulf approach.

Based on the measured computational performance of the test hardware; the time to compute a complete sonar performance prediction for a representative shallow water sonar with 72 beams out to 30Nm with 21

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE VI. Number of Beams Above Which Beowulf Clustering Should be Used

Nm	Number of Rays to be Computed per Beam												
	1	10	20	30	40	50	60	70	80	90	100	110	120
1	31	26	22	19	17	16	14	13	13	12	11	11	10
2	30	22	17	14	13	11	10	10	9	9	8	8	7
3	29	19	14	12	10	9	9	8	7	7	7	7	6
4	29	17	13	10	9	8	7	7	7	6	6	6	6
5	28	16	11	9	8	7	7	6	6	6	6	6	5
6	28	14	10	9	7	7	6	6	6	6	5	5	5
7	27	13	10	8	7	6	6	6	5	5	5	5	5
8	27	13	9	7	7	6	6	5	5	5	5	5	5
9	26	12	9	7	6	6	6	5	5	5	5	5	5
10	26	11	8	7	6	6	5	5	5	5	5	5	5
11	25	11	8	7	6	6	5	5	5	5	5	5	4
12	25	10	7	6	6	5	5	5	5	5	5	4	4
13	24	10	7	6	6	5	5	5	5	5	5	4	4
14	24	10	7	6	5	5	5	5	5	5	4	4	4
15	24	9	7	6	5	5	5	5	5	4	4	4	4
16	23	9	7	6	5	5	5	5	5	4	4	4	4
17	23	9	6	6	5	5	5	5	4	4	4	4	4
18	22	9	6	6	5	5	5	5	4	4	4	4	4
19	22	8	6	5	5	5	5	4	4	4	4	4	4
20	22	8	6	5	5	5	5	4	4	4	4	4	4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE VI. Continued

Nm	Number of Rays to be Computed per Beam												
	1	10	20	30	40	50	60	70	80	90	100	110	120
21	22	8	6	5	5	5	5	4	4	4	4	4	4
22	21	8	6	5	5	5	4	4	4	4	4	4	4
23	21	8	6	5	5	5	4	4	4	4	4	4	4
24	21	7	6	5	5	5	4	4	4	4	4	4	4
25	20	7	6	5	5	5	4	4	4	4	4	4	4
26	20	7	6	5	5	5	4	4	4	4	4	4	4
27	20	7	6	5	5	4	4	4	4	4	4	4	4
28	20	7	5	5	5	4	4	4	4	4	4	4	4
29	19	7	5	5	5	4	4	4	4	4	4	4	4
30	19	7	5	5	5	4	4	4	4	4	4	4	4

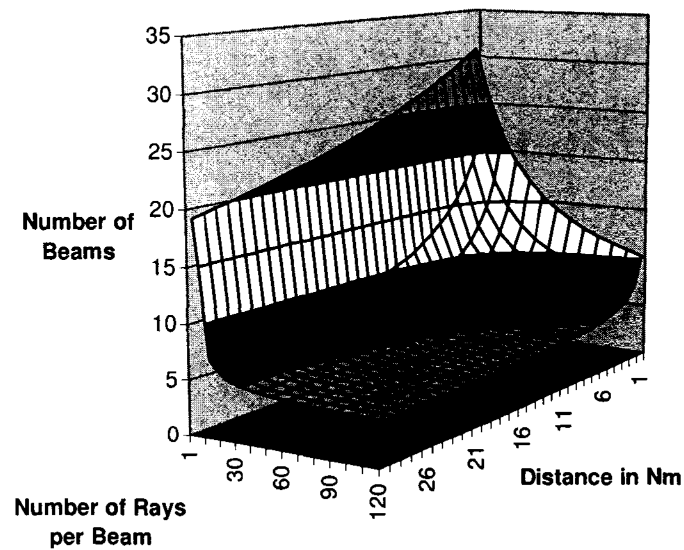


Fig. 22. Number of Beams Where Serial Out Performs Beowulf

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

rays per beam would be 22.78 seconds using a serial approach or 1.57 seconds using a Beowulf approach.

8.4 USE MEASURED DATA TO PREDICT CHANGES IN CPU AND LAN SPEEDS

Once values for the length of time that each slave took to process each ray directive and the effective LAN speed, a Beowulf simulator was written to facilitate investigating changes in CPU and LAN speeds. Appendix D contains the listing of the simulator used to explore different commercially available CPU and LAN combinations.

The model was "tuned" by varying the average program execution time and the effective LAN speed until the modeled values agreed reasonably with the measured data. Once these values were identified, the effects of changes in CPU clock and LAN speeds could be modeled. First, the execution was halved to reflect a doubling of the CPU clock speed. Increases in the CPU performance had minimal effect on the modeled curves, so CPU time was restored to the original value and changes in LAN speed were explored. Fig. 23 shows measured data and 2 simulator runs based on different network speed. The Modeled curve was used to "tune" the simulator until its shape and values approximated the measured data in the range of 3 to 7 processors. The test environment used a 100BaseTx LAN (cables, switch and NICs). The effective LAN speed was less than 100 Mbits due to signaling techniques, handshaking, system overhead, etc. Increasing the effective speed by a factor of 10 (thereby assuming that the same percentage of communications overhead exists at the new network speed) results in a much lower execution time overall. The overall shape of the modeling curves is affected by two factors, CPU speed and LAN speed. CPU speed can be seen in the major step declines in the plot. As shown when the 11th processor is brought on line on the GigaBit LAN curve. The plateaus are areas where the system is I/O bound because of the effective LAN speed. On the GigaBit curve the region from 7 to 10 processors is I/O bound, meaning that there is negligible benefit from adding the eighth, ninth, or tenth processor. If the system designer needs to have performance in the sub one second range, then 7 processors would be enough. The modeled GigaBit system has a lengthy plateau from 11 to 20 processors at slightly less than 0.5 second. The nex performance increase is at 21 processors, one processor per ray. Based on the measured data, the test environment is probably I/O bound somewhere between the third and fourth processor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

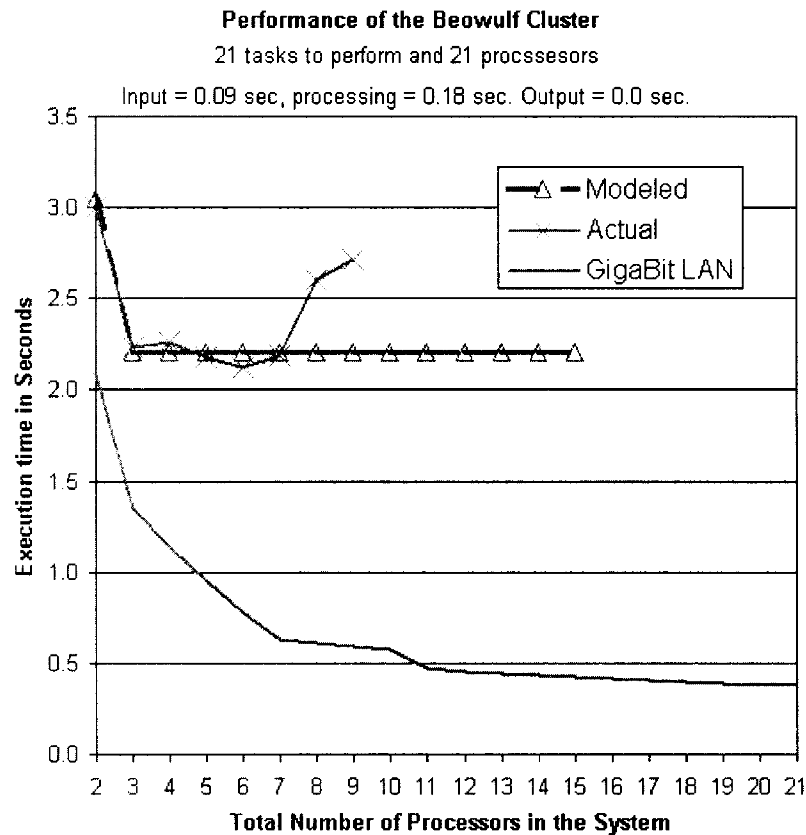


Fig. 23. Estimated Effects of Increasing LAN from 100 Mbit to 1000 Mbit

Another way to evaluate the system is to plot the effect on system performance by the addition of each processor. Figure 24 shows the effect of incrementally adding processors to the system. Using the measured data, when the third processor is added, it improves overall performance by about 26%, while the fourth adds little positive improvement, but the fifth and six each improve the system somewhat. The seventh and eighth processors actually cause the system performance to degrade. While the modeled performance for the 100 Mb LAN shows a continuous improvement for each additional processor until becoming I/O bound at 7 processors. The curves for a GigaBits LAN were not plotted because all data would have been based on conjecture.

The current work investigated whether or not it is theoretically possible to compute a sonar performance prediction in real-time using a Beowulf clustering approach. That goal has been achieved. Several areas deserve further study. They include:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

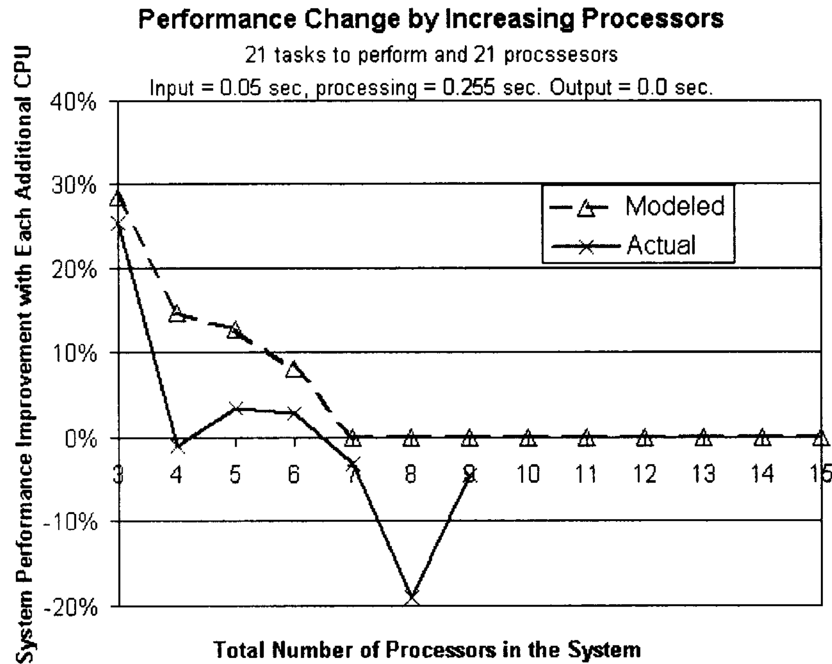


Fig. 24. Beowulf System Performance Improvement by Increasing the Number of Processors

SECTION 9. FUTURE WORK

- Expanding the single beam case to a full system. Some questions that deserve investigation include (in the case of the representative sonar, there are 72 beams), does the current approach scale up to handle that processing and communications load? Testing done during this effort identified an unexpected communications constriction when the 8th slave was added to the system (see Fig. 21). Scaling to a full 216-processor system (72 beams with 3 processors each) may reveal other unknown and unexpected restrictions.
- Determining the optimum number of processors to use. It may be possible to greatly reduce the number of required processors estimated by this paper by careful scheduling of slave utilization. This would require some degree of system control and administration whose time would have to be accounted for somewhere. Ultimately resulting in a different set of system design parameters.
- Investigate the effect of more powerful processors and faster LAN communications. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processors and network used in the test environment are not leading edge technology. Faster processors would reduce execution time of the Ray program on the individual processors, but if the LAN cannot move the data fast enough, the system would probably degenerate very quickly into an I/O bound condition.

SECTION 10. CONCLUSIONS

Real-time sonar performance prediction is possible using Beowulf clustering techniques. There is a breakeven point between serial and Beowulf computational approaches that is dependent on the number of processors and the LAN that connects them. The time to compute the total probability of detection [p(D)] for a representative shallow water sonar system is based on the following relationships:

- Serial approach = serial computation time for each beam times the number of beams
- Beowulf approach = serial computation time per beam

Based on data collected from the test hardware suite, the computational times in seconds for the serial and Beowulf approaches can be computed using the following equations:

- Serial computation time per beam = 0.0169 + number of rays * 0.00045 * distance (in Nm)
- Beowulf computation time per beam = 0.5300 + number of rays * 0.00168 * distance (in Nm)

For a representative shallow water sonar system with 72 beams, shooting 21 rays to emulate an acoustic wave front, to a range of 30 Nm; the serial approach will take 21.5 seconds and the Beowulf approach will take 1.6 seconds. The coefficients in the serial and Beowulf computation time equations show that there are system breakeven points based on the number of rays, number of beams and distance. A sonar system designer can determine the least costly and most effective hardware and software suite required to achieve real-time sonar prediction capability by using the approaches and equations derived in this paper.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

[1] J. D. Bernal, "A History of Classical Physics From Antiquity to the Quantum," Barnes & Noble Books, 1997.

[2] J. B. Bowlin, "Ocean Acoustic Ray-Tracing Software RAY WHOI-93-10," Woods Hole Oceanographic Institute, 1992.

[3] C. Cartledge, "Model 610E Mod 1 Sonar System, Signal Excess Prediction Equations," EDO Corporation Combat Systems, report 990801, 1997.

[4] C. Cartledge, "Application of an Adaptive Beowulf Computational Approach to Reduce Acoustic Environmental Modeling Times," Undersea Defence Technology Europe Conference, 2003.

[5] D. E. Culler and J.P. Singh, "Parallel Computer Architecture, A Hardware/Software Approach," Morgan Kaufmann Publishers, Inc., 1999.

[6] C. D. Hodgman, "Handbook of Chemistry and Physics," Chemical Rubber Publishing Co., 1952.

[7] R. W. Hockney, and C. R. Jesshope, "Parallel Computers," Adam Hilger Ltd, Bristol, 1981.

[8] I. Miller and J. E. Freund, "Probability and Statistics for Engineers," Prentice-Hall, Inc., 1977.

[9] A. Modi, "Real-time Visualization of Aerospace Simulations Using Computational Steering and Beowulf Clusters," 2002. Available: http://www.anirudh.net/phd/thesis.pdf.

[10] J. J. O'Connor and E. F. Robertson, "Pierre de Fermet." Available: http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Fermat.html

[11] P. S. Pachecho, "Parallel Programming with MPI," Morgan Kaufmann Publishers, Inc., 1997.

[12] E. M. Podeszwa, "Sound Speed Profiles for the North Atlantic Ocean," Naval Underwater Systems Center, 1976.

[13] G. Snow, "Underwater Acoustics and Sonar Transducers," EDO Western Division, report 17018, 1983.

[14] T. Sterling, "Beowulf Breakthroughs," Linux Magazine, June 2003. Available: http://www.linux-mag.com/2003-06/breakthroughs_01.html

[15] A. S. Tanenbaum, "Computer Networks," Prentice-Hall, Inc., 1996.

[16] TOP-500, "Top 500 Computer Sites, November 2003." Available: http://www.top500.org/

[17] R. J. Urick, "Principles of Underwater Sound for Engineers," McGraw-Hill, Inc., 1967.

[18] R. J. Urick, "Sound Propagation in the Sea," Defense Advanced Research Projects Agency, 1979.

[19] C. Wasel, "Parallel Computer Taxonomy," Aberystwyth University, 1994. Available: http://www.gigaflop.demon.co.uk/comp/chapt7.htm

[20] "NIST Handbook 148," National Institute of Standards and Technology Dataplot Reference Manual. Available: http://www.itl.nist.gov/div898/software/dataplot/refman1/ch5/weights.pdf

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[21] MathWorks, "Curve Fitting Toolbox." Available:
http://www.mathworks.com/access/helpdesk/help/toolbox/curvefit/ch_fitt5.html#67660

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX A. DEVELOPMENT OF SNELL'S LAW OF SINES

A perspective on the problems, techniques and people involved in the development of Snell's Law of Sines.

A.1 REFRACTION AND REFLECTION

Since the dawn of recorded history, man has known of the reflective properties of certain substances, for instance still water reflecting the sky. Roger Bacon's Opus Majus, in 1267 [1] is credited as one of the first Western European books including a section on the study of optics.

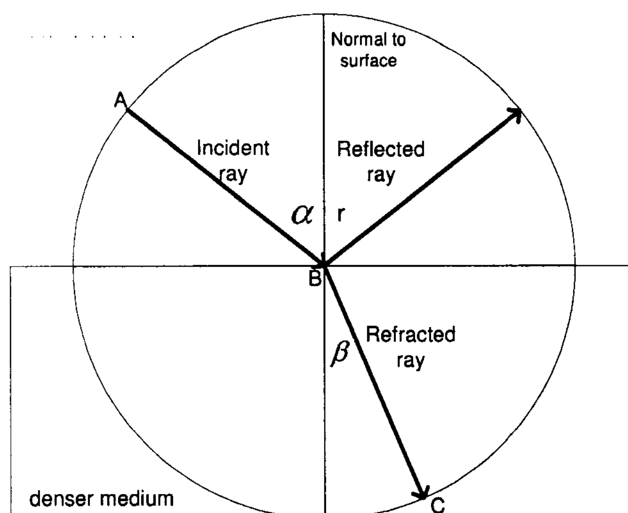


Fig. 25. Unit Circle Showing Three Standard Rays

Fig. 25 shows the major terms that will be used time and again in the following discussions. A beam of light starts at point A and hits a reflecting medium at point B. A portion of the ray reflected off of the surface and is called the reflected ray. Another portion of the ray is absorbed by the reflecting body and is called the refracted ray. The angles α and r were easily measured during Bacon's time and were exactly equal (to the limit of the measuring devices) when measured from a line normal to the surface at point B. Angles α and β could also be measured, but the relationship between the two was not known until 1703.

Willebrord Snell (1580-1626) studied law at the University of Leiden in the Netherlands, and was very interested in mathematics. He taught mathematics even while studying law. As was typical of many of the formally educated people of the 1600s, he traveled to various European countries, mostly attending lectures and discussing astronomy. In 1602 he settled in Paris where his studies continued. In 1617 Snell published

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Eratosthenes Batavus, which contains his methods for measuring the Earth, in which he proposed the method of triangulation. This work is the foundation of geodesy.

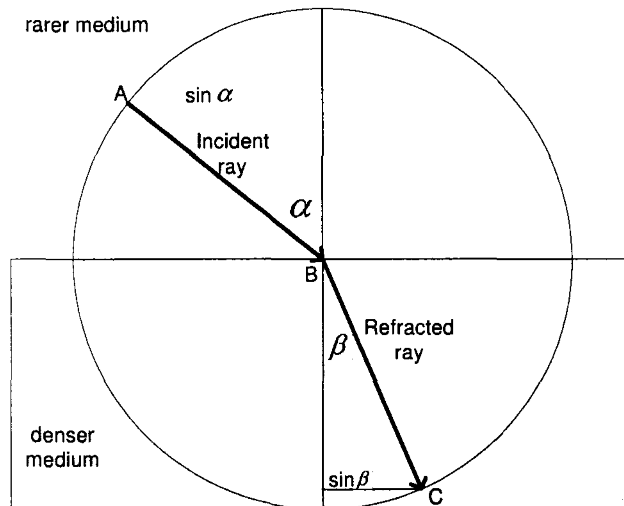


Fig. 26. Snell's Law Of Sines Diagram

$$\frac{\sin\alpha}{\sin\beta} = k \tag{9}$$

Snell's contribution to the topic of ray tracing (Appendix G) is the law of sines that bears his name. Snell's Law of Sines, as shown in Figure 26, was the discovery that the ratio of the sines of α and β were constant for any pair of mediums. Eq. (9) is Snell's Law of Sines. He was not able to explain the basis of the constant, just that it was a constant.

Pierre de Fermat (1601-1665) studied Law at the University of Orleans and had a prosperous and professionally active career in law and the judiciary. Fermat is known as a superb mathematician, even though he did not publish often. His technique of getting others to follow his line of thinking was to pose a particularly difficult question along with the answer and challenge other mathematicians to come up with the same answer. Only afterwards would he reveal his technique, often pushing the bounds of math at the time.

Fermat and Rene Descartes had a long running continuous professional and personal disagreement about Snell's Law of Sines. Neither would dispute that Snell's approach matched the measured data, what they

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

disagreed on was "why" there was a constant. Descartes took the position that light traveled faster through the denser medium than it did in the rarer medium. Fermat took the position that light was taking the least time path to get from point A to point B. It is probable that Descartes based his position on the behavior of sound. Descartes' reasoning was that sound travels faster in a denser medium compared to a rarer medium, so why shouldn't light? Fermat took the position that light traveled slower in the denser medium but that the time it took for light to travel, as shown in Fig. 27 from point A to point C via point B, was less than the time from point A to C via D.

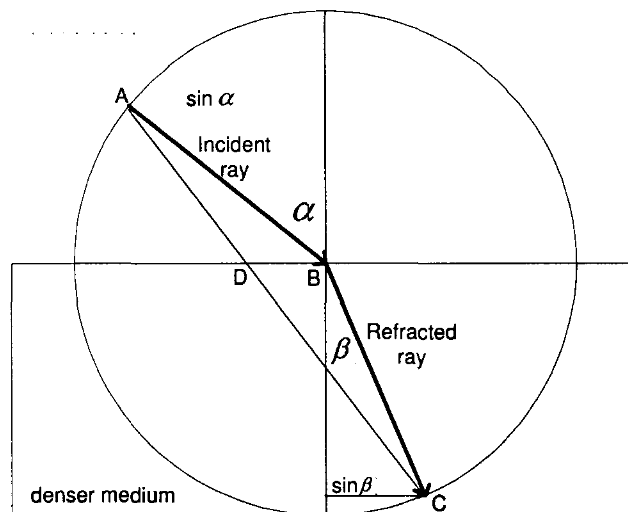


Fig. 27. Pierre De Fermat's Ray Diagram

$$|AB| = |BC| \tag{10}$$

$$|AB| + |BC| > |AC| \tag{11}$$

$$(|AD|/v1 + |DC|/v2) > (|AB|/v1 + |BC|/v2) \tag{12}$$

Equations (10) through (12) show the basis for Fermat's argument. Eq. (10) is his first assumption that the distances are equal. Because of the first assumption then the second next assumption, Eq. (11) follows. The only way that (10) and (11) could be true is if (12) was also true. Fermat could not assign values to the speed of light (v1 and v2 in the equations), he could only theorize about their relationship. Another 200 years would pass before progress could be made.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.2 WHAT IS THE SPEED OF LIGHT?

Hippolyte Fizeau (1819 - 1896) in 1849 was the first person to use terrestrial based techniques to measure the speed of light. Danish astronomer Ole Romer in 1676 had calculated the speed of light using the motion of the earth and the moons of Jupiter as 212,000 km/s (an error of 29%), but Fizeau is credited with the calculating the speed as 313,000 km/s using a device like the one shown in Fig. 28 [10].

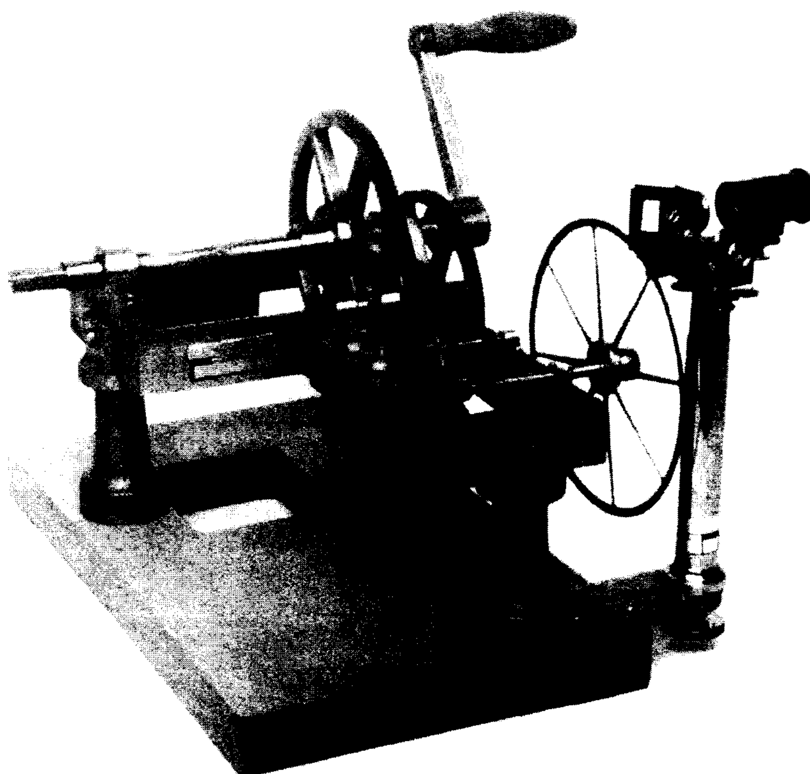


Fig. 28. Fizeau's Machine

Through a series of reduction gears, the hand crank causes the wheel in front of the eyepiece to rotate. The wheel has 720 notches cut into its outer edge. In front of the eyepiece is a partially reflecting mirror mounted so that a light source can be placed to the left of the eyepiece and focused on the mirror. The mirror reflects the light through notches in the wheel and out to a reflecting mirror some distance away. This reflected light is focused back through the eyepiece to the viewer.

In Fizeau's experiment, the mechanical device was in Mountartre, France and the reflecting mirror was on Mount Valerien in Suresnes, 8,633 meters away. The basis for measuring the speed was to cause the wheel

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to spin at just the right rate so that light passing through one notch would be able to return through the same notch. Fizeau found that the right speed was 12.6 revolutions per second. Equations (13) through (18) are similar to those that Fizeau used to compute the speed of light. Eq. (13) is the definition of the time that light took to travel from Fixeau's eyepiece and return. Eq. (14) expresses the size of one of the notches in the wheel. Eq. (15) is the angular velocity of the notch in wheel of the apparatus. Eq. (16) is the time that light would be visible through the notch. Eq. (17) sets the time for the light to transit the valley equal to the time that the light could be visible through the notch. Finally, (18) reorders the term and solves for c.

$$T1 = \frac{2*d}{c} = \frac{2*8633}{c} \tag{13}$$

$$a = \frac{2*\pi}{2*720} \tag{14}$$

$$v = 2*\pi*12.6 \tag{15}$$

$$T2 = \frac{a}{v} = \frac{\frac{2*\pi}{2*720}}{2*\pi*12.6} = \frac{1}{2*720*12.6} \tag{16}$$

$$T1 = T2 = \frac{2*d}{c} = \frac{a}{v} \tag{17}$$

$$c = \frac{2*d*v}{a} = \frac{2*8633*2*720*12.6}{1} = 313{,}000 km/s \tag{18}$$

Fizeau's measurements for the speed of light is only 4.3% greater than the 21st century accepted speed of 300,00 km/s. The size of the experimental apparatus (8,633 meters) precluded it from being used to measure the speed of light in anything other then air.

Leon Foucault (1819-1868) in 1862 was able to measure the speed of light in a laboratory. His device is shown diagrammatically in Fig. 29.

A light is focused on a rotating mirror. The rotating mirror bounces the light to a stationary mirror that reflects the light back to the rotating mirror. During this time, the mirror has rotated to a different angle,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

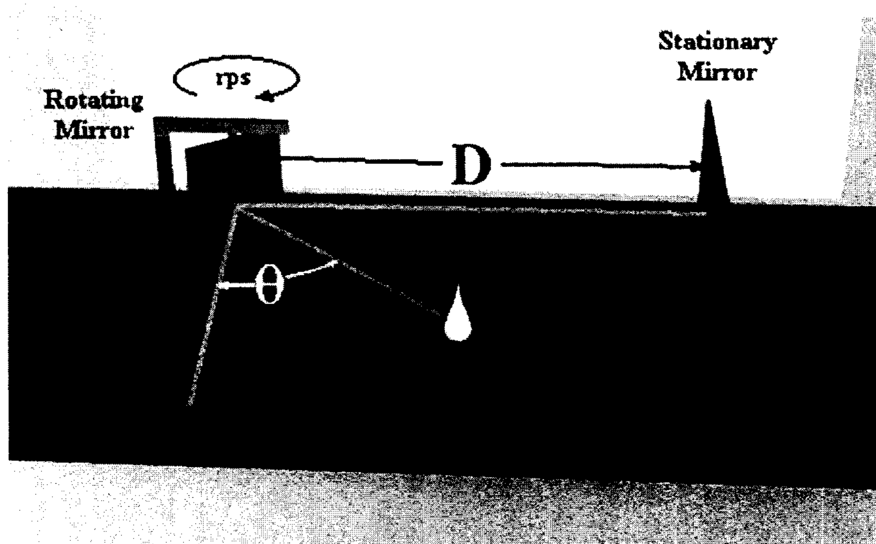


Fig. 29. Notational Diagram Of Foucault's Device To Measure The Speed Of Light

therefore the reflected light doesn't return to its source, but is displaced Θ radians away. The angular displacement Θ is related to twice the time it takes light to travel from the rotating mirror to the stationary mirror.

Foucault's experimental system permits the speed of light to be measured based on Eq. (19) through (22). Eq. (19) is the time required for light to go from the rotating mirror to the stationary mirror and then return. Eq. (20) is the time that the mirror will rotate through some number of radians of rotation. Eq. (21) sets the two times equal to each other. While Eq. (22) rearranges terms so that the speed of light can be calculated based on mechanical parameters that are under control of the experimenter.

Foucault's Eq. (22) enables the speed of light to be measured under laboratory conditions. The right hand terms are quantities that an experimenter could control based on the availability of hardware and space. TABLE VII presents representative values of θ and D showing the required mirror rotational speed in revolutions per second that would be required to compute c.

Because Foucault's apparatus used a D that was several orders of magnitude less than Fizeau's, it was possible to measure the speed of light through different substances. For example, a container filled with a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

gas could be placed in the path between the rotating and stationary mirrors and the speed of light through the gas could be measured.

$$T1 = \frac{2*D}{c} \tag{19}$$

$$T2 = \frac{1}{v} * \frac{\theta}{2*\pi} * \frac{1}{2} = \frac{\theta}{2*2*\pi*v} \tag{20}$$

$$T1 = T2 = \frac{2*D}{c} = \frac{\theta}{2*2*\pi*v} \tag{21}$$

$$c = \frac{2*2*\pi*v*2*D}{\theta} = \frac{8*\pi*v*D}{\theta} \tag{22}$$

TABLE VII. Representative Mirror Angular Velocities Based On Foucault's Apparatus

Expected Θ	D			
	1 meter	20 meters	30 meters	40 meters
0.1 degrees	21	1	1	1
0.5 degrees	104	5	4	3
1.0 degrees	208	10	7	5
1.5 degrees	312	16	10	8
2.0 degrees	417	21	14	10
2.5 degrees	521	26	17	13

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Foucault's measurements resulted in a speed of light of 304,000 km/s, within 1% of the current speed of light. Based on Foucault's apparatus, the speed of light of a particular wavelength through many different substances was measured. A wavelength of 589 mμ (yellow sodium light) is normally used as a standard for measureing indices of refraction. These measurements were normalized to the speed of light through a vacuum and are called the "index of refraction" for that substance. TABLE VIII [6] lists the index of refraction of various substances:

TABLE VIII. Index Of Refraction For Selected Substances

Substance	**Index of Refraction**
Air	1.0003
Water (H_20) at 20C	1.3330
Ice (H_20)	1.309
Turpentine	1.4721
Benzene	1.5012

Based on the various refraction indices from devices derived from Foucault's original design, the constant from Snell's Equation could be understood.

Snell's Law of Sines, Fig. 30 and Eq. (23) could now be related to the indices of refraction of the different substances that the ray was traversing through. This important relationship that is used extensively in the study of acoustic rays is that the ratio of the indices of refraction is based on the speed of something through a substance. In Snell's case, it was the speed of light. In the acoustic world, it is the speed of sound rather than light that is of importance and the speed of sound can vary within the same substance.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

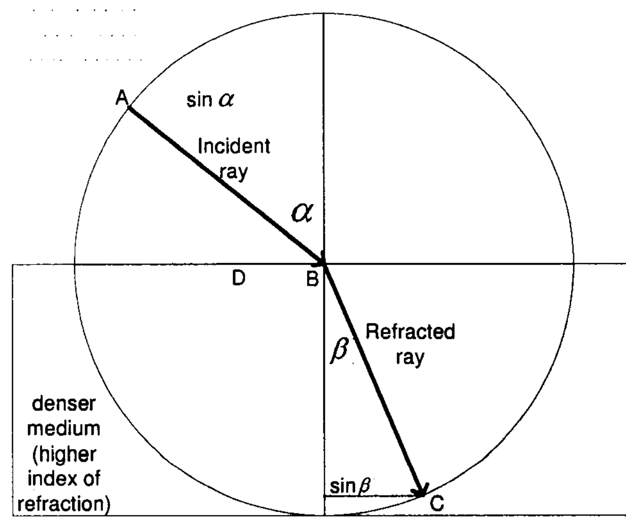


Fig. 30. Snell's Ray Diagram With Refraction Indices

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX B. SELECTED SONAR BEAM PATTERNS

B.1 GENERAL BACKGROUND INFORMATION ABOUT SONAR BEAMS

Active sonar is composed of many different mechanical and electrical components. The "working end" of the sonar is composed of transducers and their mechanical supports. A transducer is a piezoelectric device that converts electrical energy into acoustic energy (like a speaker), or converts acoustic energy into electrical energy (like a microphone). The performance of these traducers has to be known and understood so that: (1) the sonar can be designed effectively and (2) its performance predicted with a reasonable degree of confidence.

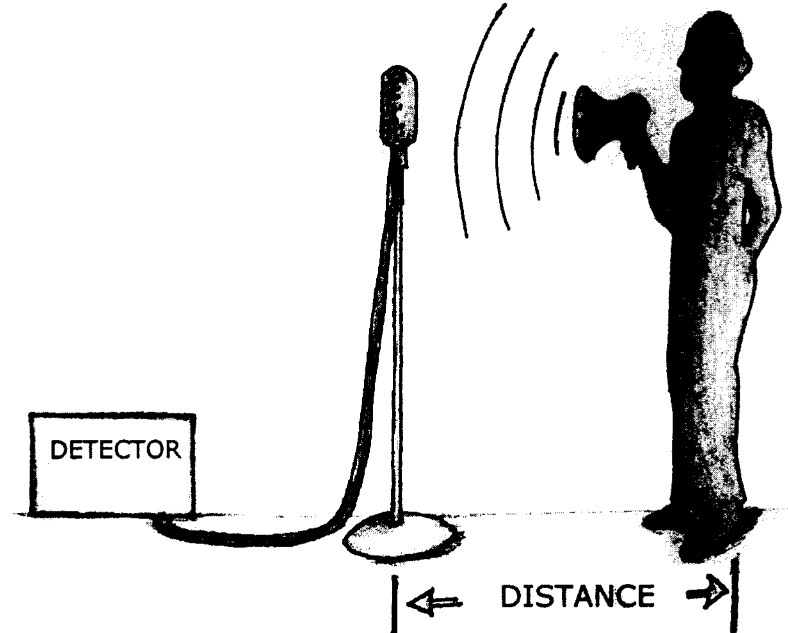


Fig. 31. Simplified Diagram Showing How The Performance Of A Microphone Is Determined

The first step in sonar design is to determine the performance of the transducer. Fig. 31 is a simplified diagram showing how a transducer's performance is measured. In the diagram a calibrated speaker is kept a fixed distance from the microphone so that the amount of acoustic energy that should be at the face of the microphone can be computed. This expected level serves as a reference level for all measurements. A detector, or recorder, or other measuring device is attached to the microphone to report the acoustic energy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

level that the microphone reports. Decibels (dB) are normally used to express the ratio of the reported and expected acoustic signals. Eq. (24) shows the definition of a decibel.

$$dB = decibel = 10 * \log_{10}(\frac{reported}{reference}) \tag{24}$$

Reporting the performance of a sonar system component in decibels has the significant advantage of being able to compute the total system performance as the sum of the decibels of each of the individual components. In Fig. 31, the human is measuring the performance of an omni directional microphone. An omni directional microphone is designed to work reasonably well through out 360 degrees of horizontal coverage. Therefore a beam pattern for that microphone might look like the one in Fig. 32 showing that the microphone is almost equally sensitive in all directions except behind it, where it is less sensitive. Omni directional microphones are designed to have nearly the same beam pattern in all directions, both horizontal and vertical. But, they are not the only types of transducers [13].

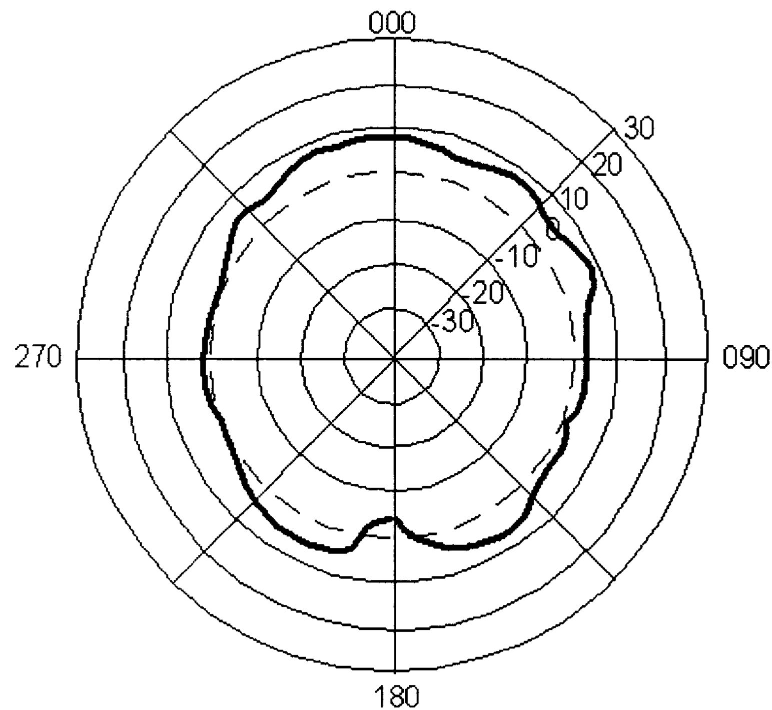


Fig. 32. Omni Directional Horizontal Beam Pattern

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B.2 TAXONOMY OF SONAR BEAMS

The mechanical construction of the piezoelectric components in a transducer has a remarkable affect on the transducer's beam pattern. TABLE IX summarizes the characteristics of several common types of transducers.

TABLE IX. Various Transducer Types And Associated Beam Patterns

Type	**Shape of piezoelectric elements**	**Shape of horizontal beam pattern**	**Shape of vertical beam pattern**
Omni directional	Spherical	Circular	Circular
Fan	Rectangular	Wide	Narrow
Toroidal	Torus	Minimal	Significant side lobes
Conical	Circular	Narrow	Narrow

Fig. 33 shows a cutaway of a conical beam transducer and its representative beam pattern. This appendix contains a selection of measured beam patterns for transducers of different shapes. The figure shows the acoustic axis of the beam and the points of maximum sensitivity and –3dB. The –3dB point is of special interest because this marks the angular point that is 50% less sensitive than the maximum. The angular measurement from the acoustic axis to the –3dB point is defined as one half the beam width. The other half of the beam width comes from the angular measurement to the –3dB point on the other side of the acoustic axis. The sum of these half beam widths is considered the beam width of the transducer. As shown in the figure, there can be multiple lower sensitivity areas/lobes; these are generically called side lobes to differentiate them from the main lobe along the acoustic axis.

Piezoelectric elements and acoustic dampeners used to construct the transducers affect the characteristic beam patterns of those transducers. This appendix contains representative beam patterns for conical, fan, omni and toroidal shaped transducers [13].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

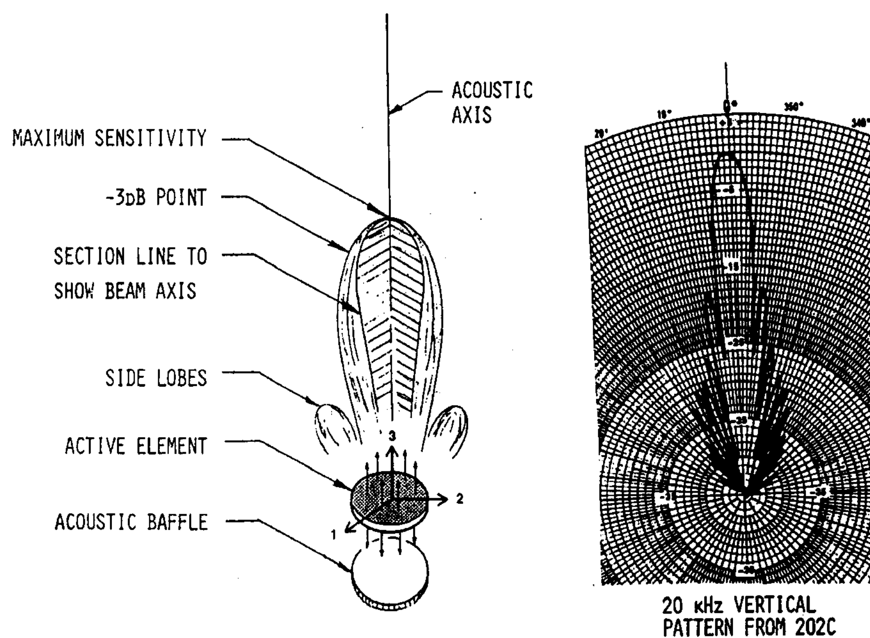


Fig. 33. Representative Beam And Beam Pattern

Conical transducers were used in the body of the paper because of their appearance to traditional audio speakers. In actuality, transducers in the representative shallow water sonar are fan shaped because that shaped transducer has minimal side lobes. When the fan shaped transducer is mounted so that the main acoustic axis is in the vertical, there is very little interference between adjacent transducers. Omni and toroidal shaped transducers have the interesting property in that they work equally well in all directions. This means that they would be able to detect something at a distance, but would not be able to tell the direction to the thing they detected. Fig. 34 through Fig. 37 are representative beam patterns for conical, fan, omni and toroidal transducers respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

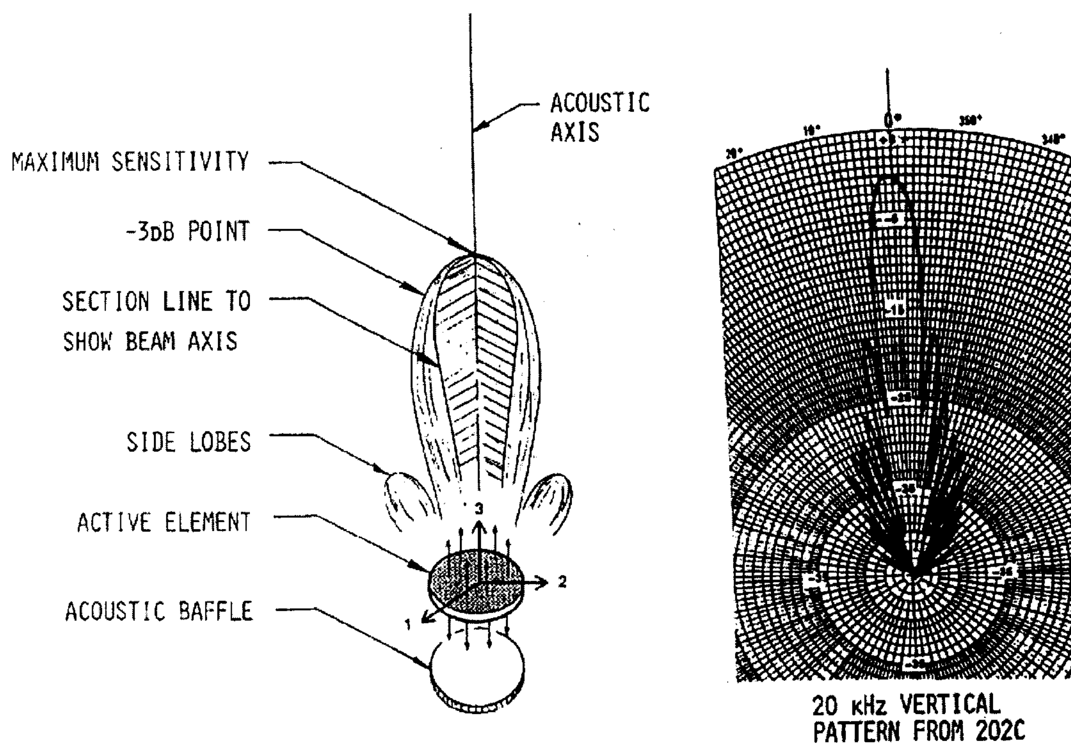


Fig. 34. Beam Pattern For A Conical Transducer

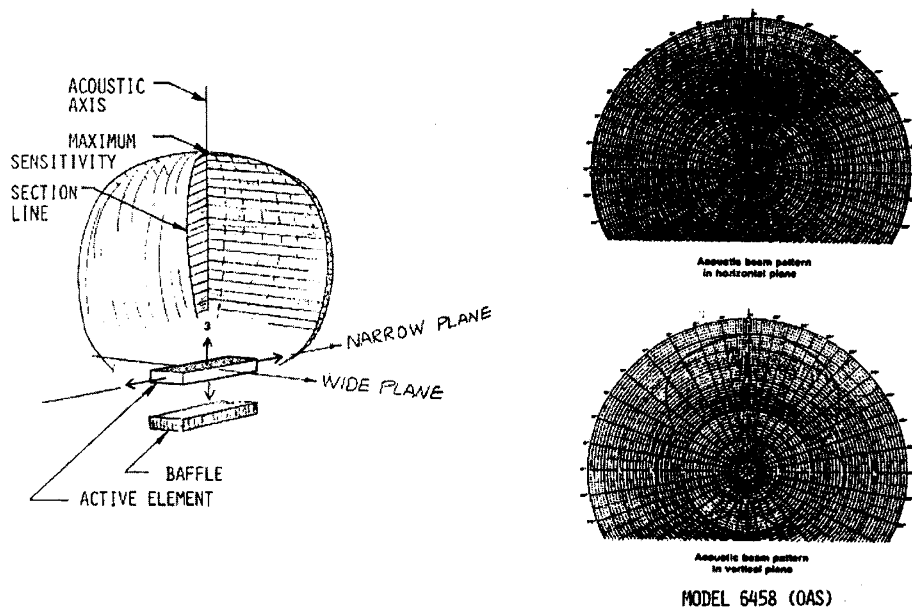


Fig. 35. Beam Pattern For A Fan Transducer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

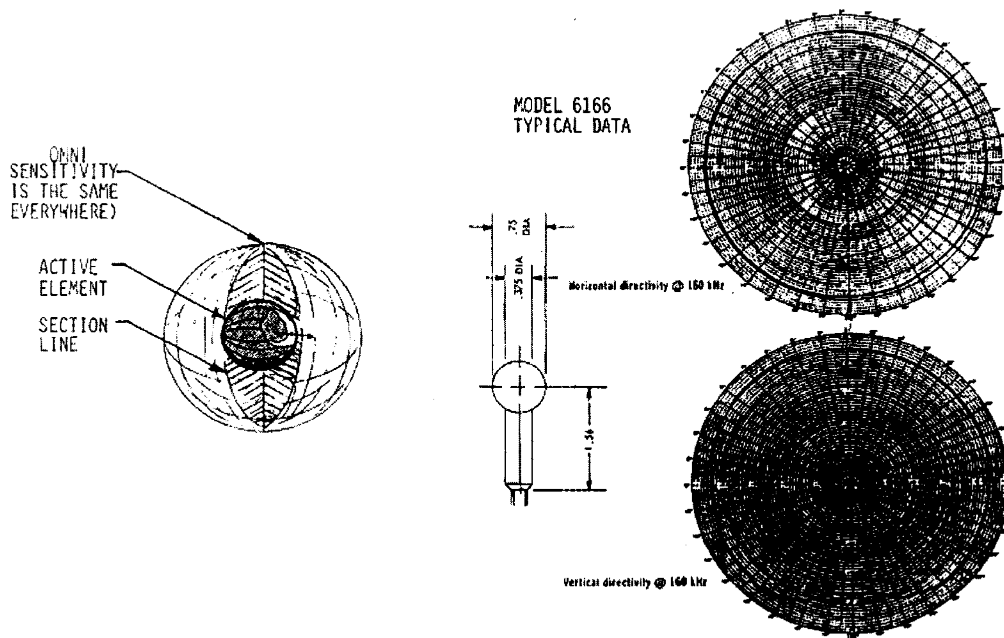


Fig. 36. Beam Pattern For An Omni Transducer

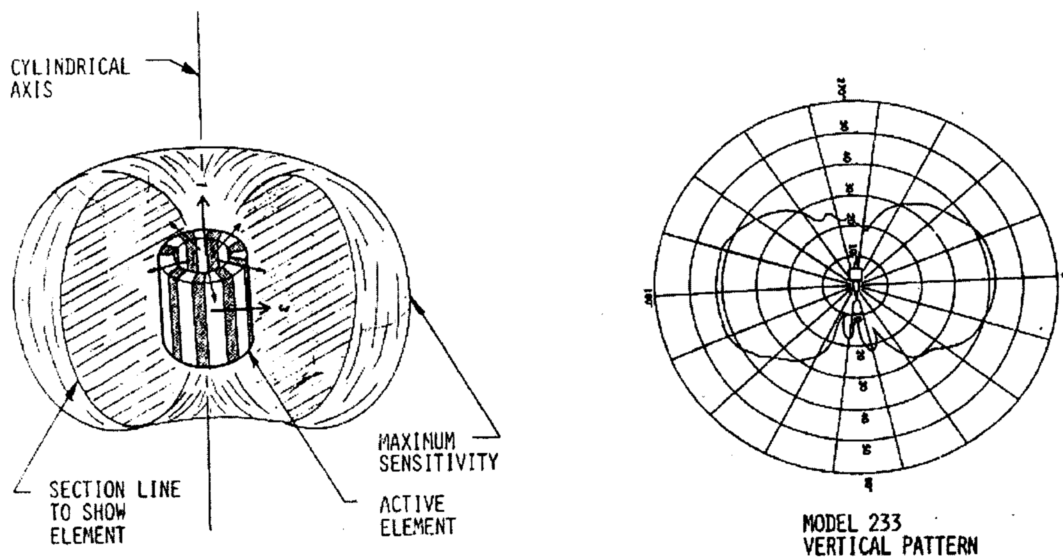


Fig. 37. Beam Pattern For A Toroidal Transducer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX C. BEOWULF (MPI) SOURCE CODE WRITTEN FOR THIS EFFORT

A complete program listing of the Beowulf master program is included in this appendix. This appendix includes the following listings:

- A makefile,
- calc.l used to parse the ray file into tokens (the file got its name from the calculator example program that served as it origin),
- calc.y used to interpret the tokens from calc.l, create individualized ray configuration files for the slave processes and to consolidate their output into a single coherent file, and
- Header files calc.h, global.h

C.1 THE MAKEFILE.

```
LEX=flex
YACC=bison
CC=mpicc
ETAGS=etags

NUM_PROCESSORS=5
NUM_BEAMS_PER_PROCESSOR=10

COLLECTION_FILE=data_collection

DEFINES=-D __USE_GNU \
        -D DEBUG_PRINT_PROGRESS_not

DEBUGS_AS_WELL = \
        -D DEBUG_ALSO

LIBS=-ll -lm

CFLAGS=$(DEFINES) $(LIBS) -ansi

BASE=calc
YACC_SRC=$(BASE).y
LEX_SRC=$(BASE).l

SRC=$(YACC_SRC) $(LEX_SRC)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
all:$(BASE)

$(BASE):$(SRC)
	$(YACC) -v -d $(YACC_SRC)
	mv $(BASE).tab.h $(BASE).h
	mv $(BASE).tab.c $(BASE).y.c
	$(LEX) $(LEX_SRC)
	mv lex.yy.c $(BASE).lex.c
	$(CC) -o $(BASE) $(BASE).lex.c $(BASE).y.c $(CFLAGS)

orig:
	$(CC) -c $(BASE).lex.c -o $(BASE).lex.o
	$(CC) -c $(BASE).y.c -o $(BASE).y.o
	$(CC) -o $(BASE) $(BASE).lex.o $(BASE).y.o $(CFLAGS)

tags:$(SRC)
	$(ETAGS) -d -t -l c $(SRC)

clean:
	rm $(BASE)

call:hello

hello:hello.c
	mpicc hello.c -o hello

run:
	mpirun -np 5 /Bshared/calc /Bshared/ray /Bshared/inputFile /BsharedBeam
#	mpirun -np 5 calc	~chuck/Ray/Source/ray1.47/ray temp~ Beam

install:
	cp calc /Bshared
	cp /home/alofts/Ray/ray1.47/ray /Bshared
	cp inputFile /Bshared
	cp test1.bth /Bshared
	cp test1.ssp /Bshared
	cp StandardRayFile.ray /Bshared
	cp master /Bshared

master:master.c
	$(CC) -o master master.c $(CFLAGS)

master_run:
	mpirun -np $(NUM_PROCESSORS) master $(NUM_BEAMS_PER_PROCESSOR) ./calc
~chuck/Ray/Source/ray1.47/ray ./BeamFiles


data:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        touch $(COLLECTION_FILE)
        ./worker $(NUM_PROCESSORS) $(NUM_BEAMS_PER_PROCESSOR) >> $(COLLECTION_FILE)

temp:
        mpirun -np 2 /Bshared/calc /Bshared/ray /Bshared/inputFile  /Bshared/beam
```

C.2 THE CALC.L FILE.

```
%{
#include "global.h"
#include "calc.h"

#include <stdlib.h>
%}

white           [ \t]+

signs            [+-]?

digit           [0-9]
intege          {digit}+
exponant        [eE]{signs}?{intege}

eal             {signs}{intege}("."{intege})?{exponant}?

text             [a-zA-Z0-9_]+

%%

{white}         { /* We ignore white space */ }

{eal}           {
                  yylval=atof(yytext);
                  return(NUMBER);
                }

"+"             return(PLUS);
"-"             return(MINUS);

"*"             return(TIMES);
"/"             return(DIVIDE);

"^"             return(POWER);

"("             return(LEFT_PARENTHESIS);
")"             return(RIGHT_PARENTHESIS);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
"\n"            return(END);

";"             return(SEMICOLON);
"{"             return(LEFT_CURLY_BRACE);
"}"             return(RIGHT_CURLY_BRACE);
"\""            return(QUOTE);
"="             return(EQUAL);

"add_extrema"   return(ADD_EXTREMA);
"min_angle"     return(MIN_ANGLE);
"iterations"    return(ITERATIONS);
"min_dz"        return(MIN_DZ);
"z_miss"        return(Z_MISS);

"angles"        return(ANGLE);
"first"         return(FIRST);
"last"          return(LAST);
"number"        return(NUMBER);
"specific"      return(SPECIFIC);
"degrees"       {strcpy (GlobalVars.temp.units, yytext); return(ANGLE_UNITS);}
"radians"       {strcpy (GlobalVars.temp.units, yytext); return(ANGLE_UNITS);}

"eigenrays"     return(EIGENRAYS);
"geo_miss"      return(GEO_MISS);
"search_max"    return(SEARCH_MAX);

"fan"           return(FAN);
"dr"            return(DR);

"input"         return(INPUT);
"prof_file"     return(PROF_FILE);
"bath_file"     return(BATH_FILE);
"loss_file"     return(LOSS_FILE);
"units"         {strcpy (GlobalVars.text, yytext); return(UNITS);}

"model"         return(MODEL);
"bathymetry"    return(BATHYMETRY);
"bath_smoothing" return(BATH_SMOOTHING);
"bottom_depth"  return(BOTTOM_DEPTH);
"bottom_type"   return(BOTTOM_TYPE);
"earth_radius"  return(EARTH_RADIUS);
"integration"   return(INTEGRATION);
"margins"       return(MARGINS);
"max_angle"     return(MAX_ANGLE);
"max_bounces"   return(MAX_BOUNCES);
"range_depend"  return(RANGE_DEPEND);
"z_tolerance"   return(Z_TOLERANCE);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
"output"        return(OUTPUT);
"ascii_file"    return(ASCII_FILE);

"paths"         return(PATHS);
"min_range"     return(MIN_RANGE);
"max_range"     return(MAX_RANGE);
"fixed_dr"      return(FIXED_DR);

"prof_smoothing" return(PROF_SMOOTHING);
"levitus"       {strcpy (GlobalVars.text, yytext); return(LEVITUS);}
"auto_max"      {strcpy (GlobalVars.text, yytext); return(AUTO_MAX);}

"source"        return(SOURCE);
"receiver"      return(RECEIVER);
"range"         return(RANGE);
"depth"         return(DEPTH);

"step_size"     return(STEP_SIZE);
"cos_factor"    return(COS_FACTOR);
"max"           return(MAX);
"min"           return(MIN);
"multiplier"    return(MULTIPLIER);

"m"             {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"km"            {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"Nmi"           {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"ft"            {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"furlong"       {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"parsec"        {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"cm"            {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"mm"            {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"Mm"            {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"lightyear"     {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}
"angstrom"      {strcpy (GlobalVars.units, yytext); return(DISTANCE_UNITS);}


{text}          {strcpy (GlobalVars.text, yytext); return(TEXT);}


<<EOF>>         return(END_OF_FILE);
%%
```

C.3 THE CALC.Y FILE.

```
%{
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#define YYDEBUG 1

#define BEGIN_FAN       "begin_fan"
#define BEGIN_WAVEFRONT "begin_wavefront"
#define DEFAULT_LENGTH  1024
#define DIE_MESSAGE     "Be gone, spawn of hell!"
#define END_FAN         "end_fan\n"
#define END_WAVEFRONT   "end_wavefront"

#if DEBUG
#define PRINT(a)  printf(a);
#else
#define PRINT(a)
#endif

#ifdef DEBUG_ALSO
#define PRINT1(v1,f1)  printf(#v1 "=%"#f1 "\n",v1)
#define PRINT2(v1,f1,v2,f2)  printf(#v1 "=%"#f1 "\t",v1); PRINT1(v2,f2)
#define PRINT3(v1,f1,v2,f2,v3,f3)  printf(#v1 "=%"#f1 "\t",v1); PRINT2(v2,f2,v3,f3)
#define PRINT4(v1,f1,v2,f2,v3,f3,v4,f4)  printf(#v1 "=%"#f1 "\t",v1);
PRINT3(v2,f2,v3,f3,v4,f4)
#else
#define PRINT1(a,f)
#define PRINT2(v1,f1,v2,f2)
#define PRINT3(v1,f1,v2,f2,v3,f3)
#define PRINT4(v1,f1,v2,f2,v3,f3,v4,f4)
#endif

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     /*  this allows us to manipulate text strings    */
#include <unistd.h>
#include "global.h"     /* this allows access to the global structures   */
#include "mpi.h"        /* this adds the MPI header files to the program */

extern FILE *yyin, *yyout;
extern char *yytext;

extern int yydebug;

extern int errno;

#define FALSE 0
#define TRUE (! FALSE)

%}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
%token NUMBER
%token PLUS   MINUS  TIMES  DIVIDE POWER
%token LEFT_PARENTHESIS     RIGHT_PARENTHESIS
%token END


%token SPECIFIC RIGHT_CURLY_BRACE LEFT_CURLY_BRACE SEMICOLON EQUAL
%token END_OF_FILE QUOTE TEXT DISTANCE_UNITS

%token ADD_EXTREMA MIN_ANGLE ITERATIONS MIN_DZ Z_MISS
%token ANGLE FIRST LAST ANGLE_UNITS
%token EIGENRAYS GEO_MISS SEARCH_MAX
%token FAN DR
%token INPUT PROF_FILE BATH_FILE LOSS_FILE UNITS
%token MODEL BATHYMETRY INTEGRATION RANGE_DEPEND BATH_SMOOTHING BOTTOM_DEPTH EARTH_RADIUS
Z_TOLERANCE
%token      MARGINS MAX_BOUNCES MAX_ANGLE BOTTOM_TYPE
%token OUTPUT ASCII_FILE
%token PATHS MIN_RANGE MAX_RANGE FIXED_DR
%token PROF_SMOOTHING LEVITUS AUTO_MAX
%token RECEIVER SOURCE RANGE DEPTH
%token STEP_SIZE COS_FACTOR MAX MIN MULTIPLIER

%left  PLUS   MINUS
%left  TIMES  DIVIDE
%left  NEG
%right POWER

%start Input
%%

Input:
         /* Empty */
       | Input Line
       ;

Line:
         END
       | END_OF_FILE                   {return;}
       | Expression END              { printf("Result: %f\n",$1); }
       | AngleLine END                 { PRINT ("Parsed something in the angles
group.\n");}
       | InputLine END                 { PRINT ("Parsed something in the input
group.\n");}
       | ModelsLine END                { PRINT ("Parsed something in the models
group.\n");}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        | OutputLine END              { PRINT ("Parsed something in the output
group.\n");}
        | SourceLine END              { PRINT ("Parsed something in the source
group.\n");}
        | StepSizeLine END            { PRINT ("Parsed something in the step size
group.\n");}
        | ReceiverLine END            { PRINT ("Parsed something in the receiver
group.\n");}
        | PathsLine END               { PRINT ("Parsed something in the path
group.\n");}
        | FanLine END                 { PRINT ("Parsed something in the fan group.\n");}
        | AddExtremaLine END          { PRINT ("Parsed something in the add extrema
group.\n");}
        | EigenRaysLine END           { PRINT ("Parsed something in the eigen
group.\n");}
        | ProfSmoothingLine END       { PRINT ("Parsed something in the prof smoothing
group.\n");}
        ;

Expression:
          NUMBER                      { $$=$1; }
        | Expression PLUS Expression  { $$=$1+$3; }
        | Expression MINUS Expression { $$=$1-$3; }
        | Expression TIMES Expression { $$=$1*$3; }
        | Expression DIVIDE Expression        { $$=$1/$3; }
        | MINUS Expression %prec NEG  { $$=-$2; }
        | Expression POWER Expression { $$=pow($1,$3); }
        | LEFT_PARENTHESIS Expression RIGHT_PARENTHESIS      { $$=$2; }
        ;


AddExtremaMinAngle:
        MIN_ANGLE AngleValue { ANGLE_VALUE(AddExtremaGroup.min_angle,"min_angle"); }
        ;

AddExtremaIterations:
        ITERATIONS EQUAL Expression SEMICOLON {
NUMERIC_VALUE(AddExtremaGroup.iterations,"iterations", $3); }
        ;

AddExtremaMinDz:
        MIN_DZ DistanceValue { DISTANCE_VALUE(AddExtremaGroup.min_dz,"min_dz"); }
        ;

AddExtremaZMiss:
        Z_MISS DistanceValue { DISTANCE_VALUE(AddExtremaGroup.z_miss,"z_miss"); }
        ;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
AddExtremaLine:
          ADD_EXTREMA AddExtremaMinAngle
        | ADD_EXTREMA AddExtremaIterations
        | ADD_EXTREMA AddExtremaMinDz
        | ADD_EXTREMA AddExtremaZMiss
          ;

AngleValue:
        EQUAL Expression AngleUnits SEMICOLON { GlobalVars.temp.value = $2; }
        ;

AngleUnits:
        LEFT_PARENTHESIS ANGLE_UNITS RIGHT_PARENTHESIS
        ;

AnglesFirst:
          FIRST AngleValue { ANGLE_VALUE(AnglesGroup.first,"first"); }
        ;

AnglesLast:
           LAST AngleValue { ANGLE_VALUE(AnglesGroup.last,"last"); }
        ;

AnglesNumber:
           NUMBER EQUAL Expression  SEMICOLON {
NUMERIC_VALUE(AnglesGroup.number,"number", $3); }
        ;

AnglesSpecific:
          ANGLE SPECIFIC EQUAL LEFT_CURLY_BRACE Expression RIGHT_CURLY_BRACE AngleUnits
SEMICOLON { $$=$1; }
        ;

AngleLine:
          ANGLE AnglesFirst
        | ANGLE AnglesLast
        | ANGLE AnglesNumber
        | ANGLE LEFT_CURLY_BRACE END AnglesFirst END AnglesLast END AnglesNumber END
RIGHT_CURLY_BRACE SEMICOLON END
        ;

DistanceUnits:
          LEFT_PARENTHESIS DISTANCE_UNITS RIGHT_PARENTHESIS {strcpy
(GlobalVars.temp.units, GlobalVars.units);}
        ;

DistanceValue:
         EQUAL Expression DistanceUnits SEMICOLON { GlobalVars.temp.value = $2; }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            ;

EigenRayGeoMiss:
        GEO_MISS DistanceValue { DISTANCE_VALUE(EigenRaysGroup.geo_miss,"geo_miss"); }
        ;

EigenSearchMax:
        SEARCH_MAX EQUAL Expression SEMICOLON { NUMERIC_VALUE(EigenRaysGroup.search_max,
"search_max", $3); }
        ;

EigenDr:
        DR DistanceValue { DISTANCE_VALUE(EigenRaysGroup.dr,"dr"); }
        ;

EigenRaysLine:
        EIGENRAYS EigenRayGeoMiss
        | EIGENRAYS EigenSearchMax
        | EIGENRAYS EigenDr
        ;

FanDr:
        DR DistanceValue { DISTANCE_VALUE(FanGroup.dr,"dr"); }
          ;

FanMaxRange:
        MAX_RANGE DistanceValue { DISTANCE_VALUE(FanGroup.max_range,"max_range"); }
            ;
FanMinRange:
        MIN_RANGE DistanceValue { DISTANCE_VALUE(FanGroup.min_range,"min_range"); }
            ;

FanLine:
          FAN FanMaxRange
        | FAN FanMinRange
        | FAN FanDr
        ;

InputBath:
          BATH_FILE EQUAL QUOTE TEXT  QUOTE SEMICOLON
{STRING_VALUE(InputGroup.bath_file,"bath_file");}
          ;

InputLoss:
          LOSS_FILE EQUAL QUOTE TEXT  QUOTE SEMICOLON
{STRING_VALUE(InputGroup.loss_file,"loss_file");}
          ;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
InputProfile:
          PROF_FILE EQUAL QUOTE TEXT  QUOTE SEMICOLON
{STRING_VALUE(InputGroup.prof_file,"prof_file");}
          ;

InputUnits:
          UNITS EQUAL QUOTE TEXT QUOTE SEMICOLON
{STRING_VALUE(InputGroup.units,"units");}
          ;

InputLine:
            INPUT InputProfile
          | INPUT InputBath
          | INPUT InputLoss
          | INPUT InputUnits
          ;

ModelsBathymetry:
            BATHYMETRY EQUAL TEXT SEMICOLON
{STRING_VALUE(ModelsGroup.bathymetry,"bathymetry");}
            ;

ModelBathSmoothing:
            BATH_SMOOTHING DistanceValue {
DISTANCE_VALUE(ModelsGroup.bath_smoothing,"bath_smoothing"); }
            ;

ModelBottomDepth:
            BOTTOM_DEPTH DistanceValue {
DISTANCE_VALUE(ModelsGroup.bottom_depth,"bottom_depth"); }
            ;

ModelBottomType:
            BOTTOM_TYPE EQUAL TEXT SEMICOLON
{STRING_VALUE(ModelsGroup.bottom_type,"bottom_type"); }
            ;

ModelEarthRadius:
            EARTH_RADIUS DistanceValue {
DISTANCE_VALUE(ModelsGroup.earth_radius,"earth_radius"); }
            ;

ModelsIntegration:
            INTEGRATION EQUAL TEXT SEMICOLON
{STRING_VALUE(ModelsGroup.integration,"integration");}
            ;

ModelMargins:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            MARGINS EQUAL Expression SEMICOLON { NUMERIC_VALUE(ModelsGroup.margins,
"margins", $3); }
            ;

ModelMaxAngle:
            MAX_ANGLE AngleValue { ANGLE_VALUE(ModelsGroup.max_angle,"max_angle"); }
            ;

ModelMaxBounces:
            MAX_BOUNCES EQUAL Expression SEMICOLON {
NUMERIC_VALUE(ModelsGroup.max_bounces, "max_bounces", $3); }
            ;

ModelsRange_depend:
            RANGE_DEPEND EQUAL TEXT SEMICOLON
{STRING_VALUE(ModelsGroup.range_depend,"range_depend");}
            ;

ModelZTolerance:
            Z_TOLERANCE DistanceValue {
DISTANCE_VALUE(ModelsGroup.z_tolerance,"z_tolerance"); }
            ;

ModelsLine:
              MODEL ModelsIntegration
            | MODEL ModelsRange_depend
            | MODEL ModelsBathymetry
            | MODEL ModelBathSmoothing
            | MODEL ModelBottomDepth
            | MODEL ModelEarthRadius
            | MODEL ModelZTolerance
            | MODEL ModelMargins
            | MODEL ModelMaxBounces
            | MODEL ModelMaxAngle
            | MODEL ModelBottomType
            ;

OutputAscii:
              ASCII_FILE EQUAL QUOTE TEXT QUOTE SEMICOLON
{STRING_VALUE(OutputGroup.ascii_file,"ascii_file");}
            ;

OutputLine:
              OUTPUT OutputAscii
              ;
PathsLine:
              MIN_RANGE DistanceValue { DISTANCE_VALUE(PathsGroup.min_range,"min_range");
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          | MAX_RANGE DistanceValue { DISTANCE_VALUE(PathsGroup.max_range,"max_range");
}
          | FIXED_DR DistanceValue { DISTANCE_VALUE(PathsGroup.fixed_dr,"fixed_dr"); }
          ;

ProfSmoothingStyle:
          LEVITUS SEMICOLON { STRING_VALUE(ProfSmoothingGroup.style,"style"); }
          | AUTO_MAX SEMICOLON { STRING_VALUE(ProfSmoothingGroup.style,"style"); }
          ;

ProfSmoothingSpecific:
          ;

ProfSmoothingLine:
          PROF_SMOOTHING ProfSmoothingStyle
          ;

SonarRange:
          RANGE DistanceValue
          ;

SonarDepth:
          DEPTH DistanceValue
          ;

ReceiverLine:
            RECEIVER SonarRange  { DISTANCE_VALUE(ReceiverGroup.range,"range"); }
          | RECEIVER SonarDepth  { DISTANCE_VALUE(ReceiverGroup.depth,"depth"); }
          ;

SourceLine:
            SOURCE SonarRange  { DISTANCE_VALUE(SourceGroup.range,"range"); }
          | SOURCE SonarDepth  { DISTANCE_VALUE(SourceGroup.depth,"depth"); }
          ;

StepSizeCosFactor:
          COS_FACTOR EQUAL Expression SEMICOLON {
NUMERIC_VALUE(StepSizeGroup.cos_factor,"cos_factor", $3); }
          ;

StepSizeMax:
          MAX DistanceValue { DISTANCE_VALUE(StepSizeGroup.max,"max"); }
          ;

StepSizeMin:
          MIN DistanceValue { DISTANCE_VALUE(StepSizeGroup.min,"min"); }
          ;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
StepSizeMultiplier:
         MULTIPLIER EQUAL Expression SEMICOLON {
NUMERIC_VALUE(StepSizeGroup.multiplier,"multiplier", $3); }
         ;

StepSizeLine:
           STEP_SIZE StepSizeMultiplier
         | STEP_SIZE StepSizeCosFactor
         | STEP_SIZE  StepSizeMax
         | STEP_SIZE  StepSizeMin
         ;

%%

           void sigcatch(int sig){
             char hostName [DEFAULT_LENGTH];
             gethostname (hostName,sizeof (hostName)-1);
             printf ("\n%s has caught signal %d and is exiting.\n\n",
                     hostName, sig);
             return (0);
           }

void outputStringData (FILE *outputFile, char *group, LineData variable){
  char tempString [DEFAULT_LENGTH];
  if (variable.used) {
    sprintf (tempString, "%s %s;\n",
             group, variable.text);
    fwrite (tempString, 1, strlen(tempString), outputFile);
  }
}

void outputQuotedStringData (FILE *outputFile, char *group, LineData variable){
  char tempString [DEFAULT_LENGTH];
  if (variable.used) {
    sprintf (tempString, "%s %s = %c%s%c;\n",
             group, variable.name,
             34,variable.text,34);
    fwrite (tempString, 1, strlen(tempString), outputFile);
  }
}

void outputNumericData (FILE *outputFile, char *group, LineData variable){
  char tempString [DEFAULT_LENGTH];
  if (variable.used){
    sprintf (tempString, "%s %s = %f",
             group, variable.name,
             variable.value, variable.units);
    if (variable.units[0]){
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      strcat (tempString," ");
      strcat (tempString,"(");
      strcat (tempString,variable.units);
      strcat (tempString,")");
    }
    strcat (tempString,";\n");
    fwrite (tempString, 1, strlen(tempString), outputFile);
  }
}

void outputIntegerData (FILE *outputFile, char *group, LineData variable){
  char tempString [DEFAULT_LENGTH];
  if (variable.used){
    sprintf (tempString, "%s %s = %d",
             group, variable.name,
             (int) variable.value);
    if (variable.units[0]){
      strcat (tempString," ");
      strcat (tempString,"(");
      strcat (tempString,variable.units);
      strcat (tempString,")");
    }
    strcat (tempString,";\n");
    fwrite (tempString, 1, strlen(tempString), outputFile);
  }
}

void appendParameterizedData(FILE *outFile){
  char tempString [DEFAULT_LENGTH];
  sprintf (tempString,"angles first = $1$ (%s);\n",
           GlobalVars.AnglesGroup.first.units);
  fwrite (tempString, 1, strlen(tempString), outFile);
  sprintf (tempString,"angles last = $1$ (%s);\n",
           GlobalVars.AnglesGroup.first.units);
  fwrite (tempString, 1, strlen(tempString), outFile);
  sprintf (tempString,"angles number = 1;\n");
  fwrite (tempString, 1, strlen(tempString), outFile);

  sprintf (tempString,"output ascii_file = \"$2$\";\n");
  fwrite (tempString, 1, strlen(tempString), outFile);
}

void createOutputFile (FILE *outFile) {
  char tempString [DEFAULT_LENGTH];

  strcpy (tempString,"input");
  outputQuotedStringData (outFile,tempString,GlobalVars.InputGroup.bath_file);
  outputQuotedStringData (outFile,tempString,GlobalVars.InputGroup.loss_file);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  outputQuotedStringData (outFile,tempString,GlobalVars.InputGroup.prof_file);
  outputQuotedStringData (outFile,tempString,GlobalVars.InputGroup.units);

  strcpy (tempString,"step_size");
  outputNumericData (outFile,tempString,GlobalVars.StepSizeGroup.max);
  outputNumericData (outFile,tempString,GlobalVars.StepSizeGroup.min);
  outputNumericData (outFile,tempString,GlobalVars.StepSizeGroup.multiplier);
  outputNumericData (outFile,tempString,GlobalVars.StepSizeGroup.cos_factor);

  strcpy (tempString,"add_extrema");
  outputNumericData (outFile,tempString,GlobalVars.AddExtremaGroup.min_angle);
  outputNumericData (outFile,tempString,GlobalVars.AddExtremaGroup.min_dz);
  outputNumericData (outFile,tempString,GlobalVars.AddExtremaGroup.z_miss);
  outputIntegerData (outFile,tempString,GlobalVars.AddExtremaGroup.iterations);

  /*  These lines are serviced in appendParameterizedData()
  strcpy (tempString,"output");
  outputQuotedStringData (outFile,tempString,GlobalVars.OutputGroup.mat_file);
  outputQuotedStringData (outFile,tempString,GlobalVars.OutputGroup.ascii_file);

  strcpy (tempString,"angles");
  outputNumericData (outFile,tempString,GlobalVars.AnglesGroup.first);
  outputNumericData (outFile,tempString,GlobalVars.AnglesGroup.last);
  outputIntegerData (outFile,tempString,GlobalVars.AnglesGroup.number);
  */
  strcpy (tempString,"eigenrays");
  outputNumericData (outFile,tempString,GlobalVars.EigenRaysGroup.geo_miss);
  outputNumericData (outFile,tempString,GlobalVars.EigenRaysGroup.dr);
  outputIntegerData (outFile,tempString,GlobalVars.EigenRaysGroup.search_max);

  strcpy (tempString,"fan");
  outputNumericData (outFile,tempString,GlobalVars.FanGroup.min_range);
  outputNumericData (outFile,tempString,GlobalVars.FanGroup.max_range);
  outputNumericData (outFile,tempString,GlobalVars.FanGroup.dr);

  strcpy (tempString,"receiver");
  outputNumericData (outFile,tempString,GlobalVars.ReceiverGroup.range);
  outputNumericData (outFile,tempString,GlobalVars.ReceiverGroup.depth);

  strcpy (tempString,"source");
  outputNumericData (outFile,tempString,GlobalVars.SourceGroup.range);
  outputNumericData (outFile,tempString,GlobalVars.SourceGroup.depth);

  strcpy (tempString,"prof_smoothing");
  outputStringData (outFile,tempString,GlobalVars.ProfSmoothingGroup.style);
}

int yywrap(){
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  int  ceaseProcessingAfterEOF = TRUE;
  /* printf ("made it to yywrap()\n"); */
  return ceaseProcessingAfterEOF;
}

void initVars(){
  memset (&GlobalVars, sizeof(GlobalVars), 0);
}

void printVars(){
  int printAddExtremaGroup = TRUE;
  int printAnglesGroup = TRUE;
  int printEigenRaysGroup = TRUE;
  int printFanGroup = TRUE;
  int printInputGroup = TRUE;
  int printModelsGroup = TRUE;
  int printfOutputGroup = TRUE;
  int printProfSmoothingGroup = TRUE;
  int printSourceGroup = TRUE;
  int printStepSizeGroup = TRUE;
  int printReceiverGroup = TRUE;

  if (printAddExtremaGroup){
    printf ("AddExtremaGroup:\n");
    PRINT_NUMERIC_DATA(AddExtremaGroup.min_angle);
    PRINT_NUMERIC_DATA(AddExtremaGroup.min_dz);
    PRINT_NUMERIC_DATA(AddExtremaGroup.z_miss);
    PRINT_NUMERIC_DATA(AddExtremaGroup.iterations);
  }

  if (printAnglesGroup){
    printf ("AnglesGroup:\n");
    PRINT_NUMERIC_DATA(AnglesGroup.first);
    PRINT_NUMERIC_DATA(AnglesGroup.last);
    PRINT_NUMERIC_DATA(AnglesGroup.number);
  }

  if (printEigenRaysGroup){
    printf ("EigenRaysGroup:\n");
    PRINT_NUMERIC_DATA(EigenRaysGroup.geo_miss);
    PRINT_NUMERIC_DATA(EigenRaysGroup.dr);
    PRINT_NUMERIC_DATA(EigenRaysGroup.search_max);
  }

  if (printFanGroup){
    printf("FanGroup:\n");
    PRINT_NUMERIC_DATA(FanGroup.max_range);
    PRINT_NUMERIC_DATA(FanGroup.min_range);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  PRINT_NUMERIC_DATA(FanGroup.dr);
}

if (printInputGroup) {
  printf ("InputGroup:\n");
  PRINT_STRING_DATA(InputGroup.prof_file);
  PRINT_STRING_DATA(InputGroup.bath_file);
  PRINT_STRING_DATA(InputGroup.loss_file);
  PRINT_STRING_DATA(InputGroup.units);
}

if (printModelsGroup) {
  printf ("ModelsGroup:\n");
  PRINT_STRING_DATA(ModelsGroup.integration);
  PRINT_STRING_DATA(ModelsGroup.range_depend);
  PRINT_STRING_DATA(ModelsGroup.bathymetry);
  PRINT_STRING_DATA(ModelsGroup.bottom_type);
}

if (printfOutputGroup){
  printf ("OutputGroup:\n");
  PRINT_STRING_DATA(OutputGroup.ascii_file);
}

if (printProfSmoothingGroup){
  printf ("Profile Smoothing Group:\n");
  PRINT_STRING_DATA(ProfSmoothingGroup.style);
}

if (printSourceGroup){
  printf ("SourceGroup:\n");
  PRINT_NUMERIC_DATA(SourceGroup.range);
  PRINT_NUMERIC_DATA(SourceGroup.depth);
}

if (printStepSizeGroup){
  printf ("Step Size Group:\n");
  PRINT_NUMERIC_DATA(StepSizeGroup.max);
  PRINT_NUMERIC_DATA(StepSizeGroup.min);
  PRINT_NUMERIC_DATA(StepSizeGroup.multiplier);
  PRINT_NUMERIC_DATA(StepSizeGroup.cos_factor);
}

if (printReceiverGroup){
  printf ("ReceiverGroup:\n");
  PRINT_NUMERIC_DATA(ReceiverGroup.range);
  PRINT_NUMERIC_DATA(ReceiverGroup.depth);
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}


int yyerror (char *msg){
  if (strcmp(yytext,"END_OF_DATA") != 0){
    printf ("yyerror() message =>%s<= yytext=>%s<=\n",msg,yytext);
  }
  return 0;
}

void postProcess(){}

void closeFile (FILE *file, char *fileName){
  char tempString[DEFAULT_LENGTH];
  if (fclose (file)){
    printf ("fclose() in closeFile() returned %d\n",errno);
  }
  if (fileName) {
    sprintf (tempString,"rm %s", fileName);
    system (tempString);
    /* printf ("%s\n",tempString); */
  }
}

int main (int argc, char *argv[]) {
  FILE *inFile;
  FILE *outFile;
  LineData firstAngle, lastAngle, number;
  double angle, angleStep;
  double *doublePtr;
  char tempFile [DEFAULT_LENGTH];
  int i, j;
  int myRank;          /* process rank */
  int processes;       /* number of processes */
  int source;          /* rank of sender */
  int dest;            /* rank of receiving process */
  int tag = 0;         /* tag for messages */
  char message[DEFAULT_LENGTH];  /* storage for message */
  MPI_Status status;  /* stores status for MPI_Recv statements */
  int workToDo;
  int workUnitsDone;
  char outputFileName [DEFAULT_LENGTH];
  char **outputFiles;
  char lineFromFile [10000];
  char tempString [DEFAULT_LENGTH];
  char *charPtr;
  int numberOfCharsRead;
  char tempRayOutputFile [DEFAULT_LENGTH];
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  int messageLength;
  char header [DEFAULT_LENGTH];
  char hostName [DEFAULT_LENGTH];
  int returnValue;


  MPI_Init(&argc, &argv); /* make MPI connection */
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank); /* finds out rank of my process */
  MPI_Comm_size(MPI_COMM_WORLD, &processes); /* finds out number of processes */

  gethostname (hostName,sizeof(hostName)-1);

  signal (SIGSEGV,sigcatch);

  returnValue = 0;
  if (myRank == 0){
    strcpy (header,"DEFAULT");
    yydebug = 0;
    /*
      for (i = 0; i < argc; i++)
      printf ("argv[%d] =>%s<=\n",i,argv[i]);
    */
    switch (argc){
    case 4:
      strcpy (header,argv[3]);
    case 3:
      inFile = fopen(argv[2],"r");
      if (inFile == 0){
       returnValue = 1;
       printf ("%s: Unable to open %s for reading.\n\texit(%d)\n",
               argv[0],argv[2],returnValue);
       goto returnExit;
      }
      yyin = inFile;
      break;
    default:
      returnValue = 3;
      printf (
              "Usage: %s LocationOfRayProgram LocationOfRayInitFile
[Header]\n\texit(%d)\n",
              argv[0],returnValue);
      goto returnExit;
    }
    initVars();
    yyparse();
    /*  printVars(); */
    closeFile (inFile, NULL);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
firstAngle = GlobalVars.AnglesGroup.first;
lastAngle = GlobalVars.AnglesGroup.last;
number = GlobalVars.AnglesGroup.number;
/* The units for the first and last angles must be made to match. */
if (strcmp(firstAngle.units, lastAngle.units) != 0){
  if (strcmp(firstAngle.units, "degrees") == 0){
   /* If this is true, then last units must be radians. */
   lastAngle.value = RADIANS_TO_DEG(lastAngle.value);
  }else{
   /* Otherwise the last units are degrees. */
   lastAngle.value = DEG_TO_RADIANS(lastAngle.value);
  }
  strcpy (lastAngle.units, firstAngle.units);
}
if (number.value == 0)
  number.value = 2;
angleStep = (lastAngle.value - firstAngle.value)/ (number.value - 1);
doublePtr =  (double *)calloc (number.value, sizeof (double));

strcpy (GlobalVars.AnglesGroup.last.units, lastAngle.units);
GlobalVars.AnglesGroup.number.value = 1;
for (i = 1, angle = firstAngle.value;
     i <= number.value;
     i++, angle += angleStep){
  *(doublePtr + (i-1)) = angle;
}

PRINT1 ("Made it here #01.",s);
sprintf (tempFile,"/Bshared/StandardRayFile.ray");
outFile = fopen(tempFile,"w+");
if (outFile  == 0){
  returnValue = 2;
  printf ("%s: Unable to open %s for output.\n\texit(%d)\n",
          argv[0],tempFile,returnValue);
  goto returnExit;
}
createOutputFile(outFile);
appendParameterizedData(outFile);
closeFile (outFile, NULL);
PRINT1 ("Made it here #02.",s);
outputFiles = (char **) calloc (number.value, sizeof(char *));

for (i = 1, dest = 1; i <= number.value; i++) {
  angle = *(doublePtr + (i-1));
  /* printf ("i = %d angle = %f %s\n",i,angle,GlobalVars.AnglesGroup.first.units); */
  sprintf (outputFileName,"%s%05d",header,i);
  *(outputFiles + (i-1)) = (char *) calloc(sizeof(outputFileName),
                                          sizeof(char));
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      strcpy (*(outputFiles + (i-1)), outputFileName);
      /*      printf ("output filename =>%s<=\n",*(outputFiles + (i-1))); */
      MPI_Send(outputFileName, strlen(outputFileName)+1,
               MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      sprintf (message,"%s %s %f %s -q >/dev/null",
               argv[1], tempFile, angle,outputFileName);
      MPI_Send(message, strlen(message)+1,
               MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      /* printf ("Sent =>%s<= to %d\n",message,dest); */
      dest ++;
      if (dest == processes)
        dest = 1;
    }

    sprintf (message,"%s", DIE_MESSAGE);
    messageLength = strlen(message)+1;
    /* Cause all the slaves to die. */
    for (dest = 1; dest < processes; dest ++)
      MPI_Send(message, messageLength,
               MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    source = 1;
    MPI_Recv(message, sizeof(message), MPI_CHAR,
             source, tag, MPI_COMM_WORLD, &status);
#ifdef DEBUG_PRINT_PROGRESS
    printf ("Process %d has completed work on %s\n",source,message);
#endif
    /* Consolidate the various output files into one. */

    outFile = fopen(GlobalVars.OutputGroup.ascii_file.text,"w+");
    /*
      printf ("opening %s for output, outFile = %x\n",
      GlobalVars.OutputGroup.ascii_file.text, outFile);
    */
    sprintf (tempRayOutputFile,"%s.asc",*(outputFiles));
    inFile = fopen(tempRayOutputFile,"r");
    /* printf ("opening %s for input, inFile = %x\n",tempRayOutputFile, inFile); */
    lineFromFile[0] = 0;
    strcpy (message,END_FAN);
    messageLength = strlen(message);

    for(workToDo = TRUE; workToDo; ){
      getline (&charPtr, &numberOfCharsRead, inFile);
      workToDo = strncmp(charPtr,message,messageLength);
      if (workToDo){
        fwrite (charPtr, 1, strlen(charPtr), outFile);
        /* printf ("writing =>%s<= %d chars\n",charPtr,strlen(charPtr)); */
      }
    }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
closeFile (inFile, tempRayOutputFile);

for (i = 2; i <= number.value; i++){
  source ++;
  if (source == processes)
   source = 1;
  MPI_Recv(message, sizeof(message), MPI_CHAR,
           source, tag, MPI_COMM_WORLD, &status);

  /* Get past the top of file info */
  sprintf (tempRayOutputFile,"%s.asc",*(outputFiles + (i-1)));
  inFile = fopen(tempRayOutputFile,"r");
  /* printf ("opening %s for input, inFile = %x\n",tempRayOutputFile, inFile); */
  lineFromFile[0] = 0;
  sprintf (tempString," %d\n",i+1);
  strcpy (message,BEGIN_FAN);
  messageLength = strlen(message);
  for(workToDo = TRUE; workToDo;
     workToDo = strncmp(charPtr,message,messageLength))
   getline (&charPtr, &numberOfCharsRead, inFile);
  /* Get past the header line */
  getline (&charPtr, &numberOfCharsRead, inFile);
  /* printf ("We are past the header stuff, now to work!\n"); */
  workToDo = TRUE;
  strcpy (message,END_FAN);
  messageLength = strlen(message);
  while (workToDo){
   getline (&charPtr, &numberOfCharsRead, inFile);
   workToDo = strncmp(charPtr,message,messageLength);
   if (workToDo){
     strcpy(lineFromFile,charPtr);
     /* printf ("processing =>%s<=",lineFromFile); */
     for (j = strlen(lineFromFile); j; j--){
       if (lineFromFile[j] == ' '){
         lineFromFile[j] = 0;
         strcat (lineFromFile,tempString);
         fwrite (lineFromFile, 1,  strlen(lineFromFile), outFile);
         break;
       }
     }
   }
  }
  closeFile (inFile, tempRayOutputFile);
}
strcpy (message,END_FAN);
messageLength = strlen(message);
fwrite (message, 1,  messageLength, outFile);
closeFile (outFile, NULL);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    postProcess();
    /* Clean up after ourselves. */
    for (i = 0; i < number.value; i++)
      free (*(outputFiles + i));
    free (outputFiles);
    free (doublePtr);
  } else {    /*  My rank is not 0, so I am a working slug. */
    workUnitsDone = 0;
    source = 0;
    dest = 0; /* sets destination for MPI_Send to process 0 */
    while (TRUE) {
      MPI_Recv(outputFileName, sizeof(outputFileName),
               MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
      PRINT2("Received #01",s,outputFileName,s);
      if (strcmp(outputFileName,DIE_MESSAGE) == 0) {
        break;
      } else {
        /* Do some work here */
#ifdef DEBUG_PRINT_PROGRESS
        printf ("Process %d (on %s) is working on %s.\n",
                myRank, hostName, outputFileName);
#endif
        MPI_Recv(message, sizeof(message), MPI_CHAR, source,
                 tag, MPI_COMM_WORLD, &status);
        PRINT2("Received #02",s,message,s);
        i = system (message);
        PRINT2("system() returned",s,i,d);
        MPI_Send(outputFileName, strlen(outputFileName)+1,
                 MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        workUnitsDone ++;
      }
    }
  }
  /* It is now time to die. */
 returnExit:
  MPI_Finalize();
  PRINT4(myRank,d,"(",s,hostName,s,") is dieing!!",s);
#ifdef DEBUG_PRINT_PROGRESS
  printf ("%s has completed its work and is dieing!\n",hostName);
#endif
  return returnValue;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C.4 THE CALC.H HEADER FILE.

Strictly speaking, this file is created by the yacc/lex processing and is included here only for completeness.

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUMBER 257
#define PLUS   258
#define MINUS  259
#define TIMES  260
#define DIVIDE 261
#define POWER  262
#define LEFT_PARENTHESIS      263
#define RIGHT_PARENTHESIS     264
#define END    265
#define SPECIFIC       266
#define RIGHT_CURLY_BRACE     267
#define LEFT_CURLY_BRACE      268
#define SEMICOLON      269
#define EQUAL  270
#define END_OF_FILE    271
#define QUOTE  272
#define TEXT   273
#define DISTANCE_UNITS 274
#define ADD_EXTREMA    275
#define MIN_ANGLE      276
#define ITERATIONS     277
#define MIN_DZ 278
#define Z_MISS 279
#define ANGLE  280
#define FIRST  281
#define LAST   282
#define ANGLE_UNITS    283
#define EIGENRAYS      284
#define GEO_MISS       285
#define SEARCH_MAX     286
#define FAN    287
#define DR     288
#define INPUT  289
#define PROF_FILE      290
#define BATH_FILE      291
#define LOSS_FILE      292
#define UNITS  293
#define MODEL  294
#define BATHYMETRY     295
#define INTEGRATION    296
#define RANGE_DEPEND   297
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#define BATH_SMOOTHING 298
#define BOTTOM_DEPTH   299
#define EARTH_RADIUS   300
#define Z_TOLERANCE    301
#define MARGINS 302
#define MAX_BOUNCES    303
#define MAX_ANGLE      304
#define BOTTOM_TYPE    305
#define OUTPUT  306
#define ASCII_FILE     307
#define PATHS   308
#define MIN_RANGE      309
#define MAX_RANGE      310
#define FIXED_DR       311
#define PROF_SMOOTHING 312
#define LEVITUS 313
#define AUTO_MAX       314
#define RECEIVER       315
#define SOURCE  316
#define RANGE   317
#define DEPTH   318
#define STEP_SIZE      319
#define COS_FACTOR     320
#define MAX     321
#define MIN     322
#define MULTIPLIER     323
#define NEG     324


extern YYSTYPE yylval;
```

C.5 THE GLOBAL.H HEADER FILE.

```
#define YYSTYPE double
#define M_PI          3.14159265358979323846264338327950 29L  /* pi */

/* Be very carefull with the next few lines.  There is magic in them. */
#define DISTANCE_VALUE(a,b) {GlobalVars.a.value = GlobalVars.temp.value; \
strcpy (GlobalVars.a.units, GlobalVars.temp.units); \
GlobalVars.a.used = TRUE; \
strcpy(GlobalVars.a.name, b);}

#define PRINT_NUMERIC_DATA(a) \
printf ("\t%s = %f (%s) (%sset)\n", GlobalVars.a.name, \
GlobalVars.a.value, GlobalVars.a.units, (GlobalVars.a.used ?"":"not "));
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#define PRINT_STRING_DATA(a) \
printf ("\t%s = =>%s<= (%sset)\n", GlobalVars.a.name, \
GlobalVars.a.text, (GlobalVars.a.used ?"":"not "));

#define NUMERIC_VALUE(a,b,c) \
{strcpy(GlobalVars.a.name, b); GlobalVars.a.value = c;GlobalVars.a.used = TRUE;}

#define STRING_VALUE(a,b) \
{strcpy(GlobalVars.a.name, b); strcpy(GlobalVars.a.text, GlobalVars.text);
GlobalVars.a.used = TRUE;}

/* The magic has ended.  Can you find it??  Have fun.  C. Cartledge, July 2003 */

#define ANGLE_VALUE(a,b) DISTANCE_VALUE(a,b);
#define DEG_TO_RADIANS(a) (M_PI/180. *(a))
#define RADIANS_TO_DEG(a) (180./M_PI *(a))

extern YYSTYPE yylval;

typedef struct lineData {
  double value;
  char units [100];
  int used;
  char name[100];
  char text[100];
} LineData;

struct {
  char units [100];
  char text [100];
  LineData temp;

  struct {
    LineData min_angle, min_dz, z_miss, iterations;
  } AddExtremaGroup;

  struct {
    LineData first, last, number;
  } AnglesGroup;

  struct {
    LineData geo_miss, dr,  search_max;
  } EigenRaysGroup;

  struct {
    LineData min_range, max_range, dr;
    int include_bounces;
  } FanGroup;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
struct {
  LineData bath_file, loss_file, prof_file, units;
} InputGroup;

struct {
  LineData integration;
  LineData range_depend;
  LineData bathymetry;
  struct {
    LineData max, min;
  } range_step;
  struct {
    LineData factor;
    LineData iteration;
  } debias;
  LineData bath_smoothing;
  LineData bottom_depth;
  LineData earth_radius;
  LineData z_tolerance;
  LineData margins;
  LineData max_bounces;
  LineData max_angle;
  LineData bottom_type;
} ModelsGroup;

struct {
  LineData mat_file;
  LineData ascii_file;
  int initialization;
  int filenames;
  int sound_speeds;
  int bathymetry;
  int fan;
  int wavefront;
  int eigenrays;
  int everything;
  int environment_only;
  int turning_points;
} OutputGroup;

struct {
  LineData min_range, max_range, fixed_dr;
  struct {
    int range;
    int depth;
    int time;
    int angle;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      int speed;
      int grad;
      int top_bounces;
      int bot_bounces;
      int ray_number;
      int everything;
    } columns;
  } PathsGroup;

  struct {
    LineData style;
  } ProfSmoothingGroup;

  struct {
    LineData range;
    LineData depth;
  } ReceiverGroup, SourceGroup;

  struct {
    LineData max, min,  multiplier, cos_factor;
  } StepSizeGroup;

} GlobalVars;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX D. SOURCE CODE FOR JAVA BASED BEOWULF CLUSTER PERFORMANCE ESTIMATOR

A Java based Beowulf simulator was written to investigate the effects of different combinations of CPU and LAN speeds and number of slave processors. At the core of a Beowulf cluster is the number of processors that are available, how fast they can talk to each other (the effective LAN speed) and the time that each processor takes to complete its assigned task. As shown in Fig. 38, the operator can enter:

- The number of available processors. These are the slave nodes because it is assumed that the master node is dedicated to purely scheduling functions.
- The effective LAN speed. The effective speed is not the advertised LAN speed. Any LAN communication has some amount of overhead that reduces the LAN speed from the theoretical value to an effective one. This entry is the effective (and ideally, the measured) value.
- The number of tasks that the slave processors are to execute. In general, the number of tasks does not have to match the number of processors.
- Each processor requires some amount of input data, so there is a way to enter that value.
- Execution of the task takes some amount of time and the operator is asked to enter the expected duration of the task.
- A task is expected to output some amount of data to a file or sent to another process, so an entry panel is available for that purpose.

When all data have been entered, the operator presses the "Start the Simulation" button and the numeric results of the simulation are presented in the Performance panel.

The simulator applies a simple model to Beowulf system execution, ignoring the effects of OS limitations and other concurrent programs. Internally the simulation builds a queue of events intended to keep the slave processors as busy as possible. To that end, a slave can be in any one of five states:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

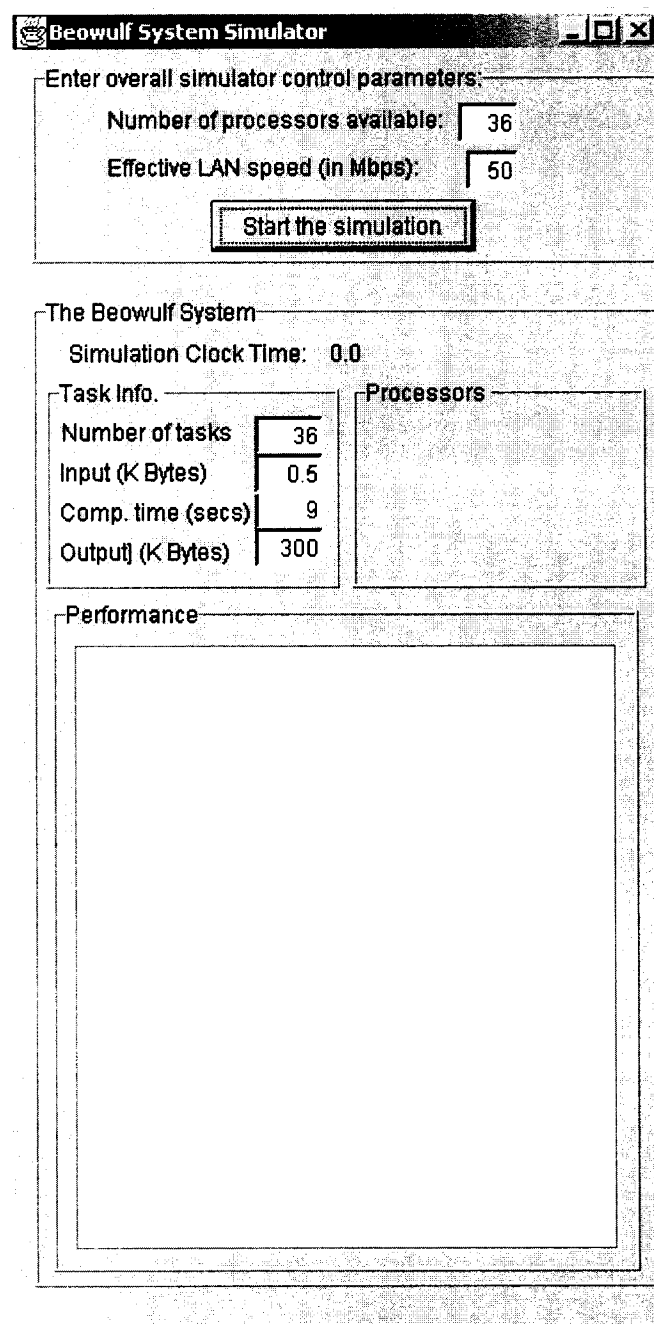


Fig. 38. Beowulf System Simulator Control Panel

1. Idle (if there is no work to be done),

2. Input (if there is data available from the master),

3. Computing (simulating doing work),

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Waiting to output data, or

5. Output (returning data to the master).

Because each of these operations takes a finite amount of time, as determined by the operator entered data, a time line of events can be constructed to mimic the expected Beowulf system operation. The simulator then models the system using a different number of processors per iteration and returns the total run time based on the number of processors. Once the model has been "tuned" to reasonably match real data, the computation time can to be changed to see the gross effects of changing CPU clock speed. Or, the speed of the LAN could be changed to see the effects of different networking techniques. By seeing the effects of these changes, a system designer can focus in on the hardware combination that will meet the system throughput requirements.

There are three files in this appendix:

1. BeowulfSystem.java – the main context for the simulator

2. Frame2.java – the operator interface and associated GUI functions

3. BeowulfProcessor.java – the Beowulf processor(s)

D.1 PACKAGE BEOWULFSIMULATOR;

```
import javax.swing.UIManager;

public class BeowulfSystem {
  boolean packFrame = false;

  //Construct the application
  public BeowulfSystem() {
    Frame2 frame = new Frame2();
    //Validate frames that have preset sizes
    //Pack frames that have useful preferred size info, e.g. from their layout
    if (packFrame) {
      frame.pack();
    }
    else {
      frame.validate();
    }
    frame.setVisible(true);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  }

  //Main method
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    new BeowulfSystem();
  }
}
```

D.2 PACKAGE BEOWULFSIMULATOR;

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.borland.jbcl.layout.*;
import javax.swing.border.*;

public class Frame2 extends JFrame {
  JPanel contentPane;
  XYLayout xYLayout1 = new XYLayout();
  JPanel jPanel1 = new JPanel();
  JLabel jLabel1 = new JLabel();
  XYLayout xYLayout2 = new XYLayout();
  JTextField numberOfProcssors = new JTextField();
  JLabel jLabel2 = new JLabel();
  JTextField effectiveLANSpeed = new JTextField();
  JButton startButton = new JButton();
  Border border1;
  TitledBorder titledBorder1;
  JPanel systemPanel = new JPanel();
  XYLayout xYLayout3 = new XYLayout();
  Border border2;
  TitledBorder titledBorder2;
  JPanel taskInfo = new JPanel();
  Border border3;
  TitledBorder titledBorder3;
  JPanel outputInfo = new JPanel();
  Border border4;
  TitledBorder titledBorder4;
  static JPanel processors = new JPanel();
  Border border5;
  TitledBorder titledBorder5;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  XYLayout xYLayout4 = new XYLayout();
  XYLayout xYLayout5 = new XYLayout();
  JLabel jLabel4 = new JLabel();
  JLabel jLabel5 = new JLabel();
  JLabel jLabel6 = new JLabel();
  JLabel jLabel7 = new JLabel();
  JTextField numberTasks = new JTextField();
  JTextField inputKbytes = new JTextField();
  JTextField compTime = new JTextField();
  JTextField outputKbytes = new JTextField();
  JScrollPane outputPanel = new JScrollPane();
  JTextArea outputArea = new JTextArea();
  JLabel jLabel8 = new JLabel();
  JLabel simulationClockTimeLabel = new JLabel();
  XYLayout xYLayout6 = new XYLayout();


  double clockTime = 0;

  //Construct the frame
  public Frame2() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  //Component initialization
  private void jbInit() throws Exception  {
    contentPane = (JPanel) this.getContentPane();
    border1 = BorderFactory.createEmptyBorder();
    titledBorder1 = new TitledBorder(new
EtchedBorder(EtchedBorder.RAISED,Color.white,new Color(142, 142, 142)),"Enter overall
simulator control parameters:");
    border2 = BorderFactory.createEmptyBorder();
    titledBorder2 = new TitledBorder(new
EtchedBorder(EtchedBorder.RAISED,Color.white,new Color(142, 142, 142)),"The Beowulf
System");
    border3 = new EtchedBorder(EtchedBorder.RAISED,Color.white,new Color(142, 142,
142));
    titledBorder3 = new TitledBorder(new
EtchedBorder(EtchedBorder.RAISED,Color.white,new Color(142, 142, 142)),"Task Info. ");
    border4 = new EtchedBorder(EtchedBorder.RAISED,Color.white,new Color(142, 142,
142));
    titledBorder4 = new TitledBorder(border4,"Performance");
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    border5 = new EtchedBorder(EtchedBorder.RAISED,Color.white,new Color(142, 142,
142));
    titledBorder5 = new TitledBorder(border5,"Processors ");
    contentPane.setLayout(xYLayout1);
    this.setSize(new Dimension(351, 700));
    this.setTitle("Beowulf System Simulator");
    jLabel1.setText("Number of processors available:");
    jPanel1.setLayout(xYLayout2);
    numberOfProcssors.setText("36");
    numberOfProcssors.setHorizontalAlignment(SwingConstants.RIGHT);
    jLabel2.setText("Effective LAN speed (in Mbps):");
    effectiveLANSpeed.setText("50");
    effectiveLANSpeed.setHorizontalAlignment(SwingConstants.RIGHT);
    startButton.setText("Start the simulation");
    startButton.addActionListener(new java.awt.event.ActionListener() {

      public void actionPerformed(ActionEvent e) {
        startButton_actionPerformed(e);
      }
    });
    jPanel1.setBorder(titledBorder1);
    systemPanel.setLayout(xYLayout3);
    systemPanel.setBorder(titledBorder2);
    systemPanel.setMinimumSize(new Dimension(800, 1000));
    contentPane.setMinimumSize(new Dimension(800, 1000));
    contentPane.setPreferredSize(new Dimension(800, 1000));
    taskInfo.setBorder(titledBorder3);
    taskInfo.setLayout(xYLayout4);
    outputInfo.setBorder(titledBorder4);
    outputInfo.setLayout(xYLayout5);
    processors.setBorder(titledBorder5);
    processors.setLayout(xYLayout6);
    jLabel4.setText("Number of tasks");
    jLabel5.setText("Input (K Bytes)");
    jLabel6.setText("Output] (K Bytes)");
    jLabel7.setText("Comp. time (secs)");
    numberTasks.setText("36");
    numberTasks.setHorizontalAlignment(SwingConstants.RIGHT);
    inputKbytes.setText("0.5");
    inputKbytes.setHorizontalAlignment(SwingConstants.RIGHT);
    compTime.setText("9");
    compTime.setHorizontalAlignment(SwingConstants.RIGHT);
    outputKbytes.setText("300");
    outputKbytes.setHorizontalAlignment(SwingConstants.RIGHT);
    jLabel8.setText("Simulation Clock Time: ");
    simulationClockTimeLabel.setToolTipText("");
    simulationClockTimeLabel.setText("0.0");
    contentPane.add(jPanel1, new XYConstraints(8, 5, 339, 109));
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    jPanel1.add(jLabel1, new XYConstraints(36, 1, 190, -1));
    jPanel1.add(numberOfProcssors, new XYConstraints(223, 1, 32, -1));
    jPanel1.add(jLabel2, new XYConstraints(36, 26, -1, -1));
    jPanel1.add(effectiveLANSpeed, new XYConstraints(227, 26, 28, -1));
    jPanel1.add(startButton, new XYConstraints(91, 52, -1, -1));
    contentPane.add(systemPanel, new XYConstraints(8, 128, 339, 527));
    systemPanel.add(jLabel8, new XYConstraints(15, 1, -1, -1));
    systemPanel.add(simulationClockTimeLabel, new XYConstraints(154, 1, 95, -1));
    systemPanel.add(taskInfo, new XYConstraints(1, 21, 160, 115));
    taskInfo.add(jLabel5, new XYConstraints(3, 21, -1, -1));
    taskInfo.add(jLabel7, new XYConstraints(3, 42, -1, -1));
    taskInfo.add(jLabel6, new XYConstraints(3, 63, -1, -1));
    taskInfo.add(numberTasks, new XYConstraints(107, 0, 37, -1));
    taskInfo.add(compTime, new XYConstraints(107, 39, 37, -1));
    taskInfo.add(outputKbytes, new XYConstraints(107, 59, 37, -1));
    taskInfo.add(inputKbytes, new XYConstraints(107, 20, 37, -1));
    taskInfo.add(jLabel4, new XYConstraints(4, 0, 98, -1));
    systemPanel.add(processors, new XYConstraints(164, 21, 160, 115));
    systemPanel.add(outputInfo, new XYConstraints(4, 138, 316, 359));
    outputInfo.add(outputPanel, new XYConstraints(8, 4, 289, 320));
    outputPanel.getViewport().add(outputArea, null);
  }

  //Overridden so we can exit when window is closed
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0);
    }
  }


  void setClocktime(double newClockTime){
    clockTime = newClockTime;
    displayClockTime();
  }

  void incrementClockTime(double increment){
    double tempDouble = clockTime;
    setClocktime(tempDouble + increment);
  }

  void displayClockTime(){
    simulationClockTimeLabel.setText(new String(Double.toString(clockTime)));
  }

  void startButton_actionPerformed(ActionEvent e) {
    int xLocationStep = 30;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int yLocationStep = 30;
int xStartLocation = 0;
int yStartLocation = 0;
int xLocation;
int yLocation;
int processorNumber;
int processorLimit;
int i;
int processor;
int task;
int taskLimit;
double nextEventTime;
boolean foundWork;
double inputTimeDuration;
double outputTimeDuration;
double taskDuration;

processorLimit = Integer.parseInt(numberOfProcssors.getText());
taskLimit = Integer.parseInt(numberTasks.getText());

BeowulfProcessor processorArray[] = new BeowulfProcessor[processorLimit + 1];

// Create the processor display
xLocation = xStartLocation;
yLocation = yStartLocation;
i = 0;
for (processorNumber = 0; processorNumber < processorLimit; processorNumber ++){
  processorArray[processorNumber] =
        new BeowulfProcessor(processorNumber,xLocation, yLocation);
        processors.add(processorArray[processorNumber],
            new XYConstraints(xLocation, yLocation, -1, -1));
               // this.setSize(15,10);
  processorArray[processorNumber].SetProcessorDurations(i,0,0,0);
  if (i == 3){
    xLocation = xStartLocation;
    yLocation += yLocationStep;
    i = 0;
  }else{
    xLocation += xLocationStep;
    i ++;
  }
}
// Processor display now complete.

// Run complete tasks list against each possible processor combination

// Convert KBytes to seconds on a Mb rated network.
inputTimeDuration = Double.parseDouble(inputKbytes.getText())
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
         * 8. / 1000. / Double.parseDouble(effectiveLANSpeed.getText());
    outputTimeDuration = Double.parseDouble(outputKbytes.getText())
         * 8. / 1000. / Double.parseDouble(effectiveLANSpeed.getText());
    taskDuration = Double.parseDouble(compTime.getText());
    outputArea.append(new String("Don't Remove!! There are this many "+taskLimit+"
tasks.\n"));
    outputArea.append(new String("Don't Remove!! Processing takes this long: " +
taskDuration + " seconds.\n"));
    outputArea.append(new String("Don't Remove!! Output takes this long: " +
outputTimeDuration + " seconds.\n"));
    outputArea.append(new String("Don't Remove!! Input takes this long: " +
inputTimeDuration + " seconds.\n"));
    outputArea.append(new String("Don't Remove!! There are barely, just " +
processorLimit + " processors\n"));
    outputArea.append(new String("Don't Remove!!\n"));
    for (processor=0; processor < processorLimit; processor++){
      outputArea.append(new String((processor + 1)
              +" processors and " + taskLimit + " tasks "));
      setClocktime(0);
      nextEventTime = 0;
      foundWork = true;
      task = 0;
      while (foundWork){
          foundWork = false;
          displayClockTime();
        /*  Look for a processor that is (in priority order):
            1.  Waiting to output data
            2.  Available to accept data
            3.  Last case, set the clock to the next event in the system
        */
        // Check for someone waiting to output data
        for (i = 0; i <= processor; i++){
          if ((processorArray[i].GetProcessorNextEventTime() <= clockTime) &&
              (processorArray[i].GetProcessorState() ==
BeowulfProcessor.WaitingOutput)){
            // Allow time for the output to occur
              // outputArea.append(new String("processor #"+i+" waitingoutput\n"));
              incrementClockTime(outputTimeDuration);
              processorArray[i].SetProcessorClockTime(0);
              processorArray[i].SetProcessorState(BeowulfProcessor.AwaitingData);
              //foundWork = true;
              break;
          }
        }
        // Check for someone waiting to accept data
        if (foundWork == false){
          if (task < taskLimit){
          for (i = 0; i <= processor; i++){
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            if ((processorArray[i].GetProcessorNextEventTime() <= clockTime) &&
                (processorArray[i].GetProcessorState() ==
BeowulfProcessor.AwaitingData)){
              // Allow time for the input to occur
                  processorArray[i].SetProcessorDurations(task,
                        inputTimeDuration, taskDuration, outputTimeDuration);
                  processorArray[i].SetProcessorClockTime(clockTime);
                  processorArray[i].SetProcessorState(BeowulfProcessor.WaitingOutput);
                  processorArray[i].ComputeNextEventTime();
                  incrementClockTime(inputTimeDuration);
                  task ++;
/*
                  outputArea.append(new String("processor #"+i
                        +" inputTimeDuration " + inputTimeDuration
                        +" NextEventTime = " +
processorArray[i].GetProcessorNextEventTime()
                        +" state = " + processorArray[i].GetProcessorState()
                        +" clockTime = " + clockTime
                        +"      \n"));
*/
                  foundWork = true;
                  break;
            }
          }
          }
        }
        // Check get the time of the next processor event
        if (foundWork == false){
          double timeNextProcessorEvent = Double.MAX_VALUE;
          double tempDouble;
          // Look for the next closest event in time
          for (i = 0; i <= processor; i++){
            tempDouble = processorArray[i].GetProcessorNextEventTime();
            if (tempDouble > 0) // There is work to be done
              timeNextProcessorEvent = Math.min (tempDouble, timeNextProcessorEvent);
          }
          if (timeNextProcessorEvent != Double.MAX_VALUE){
            // Only happen if there is work to be done
            if (clockTime < timeNextProcessorEvent)
                setClocktime(timeNextProcessorEvent);
              foundWork = true;
            }
        }

        // At the end of searching.  If foundWork == false, at the all work
        }
        outputArea.append("takes " + clockTime +" seconds.\n");
      }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  }
}

/**
 * Title:        <p>
 * Description:  <p>
 * Copyright:    Copyright (c) <p>
 * Company:      <p>
 * @author
 * @version 1.0
 */
```

D.3 PACKAGE BEOWULFSIMULATOR;

```
import java.awt.Color;

public class BeowulfProcessor extends javax.swing.JButton{

  static int AwaitingData = 0;
  static int RecvingInput = AwaitingData +1;
  static int Processing = AwaitingData + 2;
  static int WaitingOutput = AwaitingData + 3;
  static int OutputingData = AwaitingData + 4;

  int ProcessorState;
  int ProcessorNumber;
  int taskNumber;

  double clockTime = 0;
  double inputTimeDuration = 0;
  double processTimeDuration = 0;
  double outputTimeDuration = 0;

  public BeowulfProcessor(int processorNumber, int x, int y) {
    ProcessorState = AwaitingData;
    ProcessorNumber = processorNumber;
    // Frame2.processors.add(this);
    // this.setLocation(x,y);
    this.setText(LableString());
    // this.setSize(15,10);
    this.setVisible(true);
  }
  void ComputeNextEventTime(){
    if (ProcessorState == AwaitingData){
      clockTime = 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      } else {
        if ((ProcessorState == RecvingInput)
         || (ProcessorState == Processing)
         || (ProcessorState == WaitingOutput)){
            clockTime += (inputTimeDuration + processTimeDuration);
        } else {
          if (ProcessorState == OutputingData){
            /* do nothing */
          }
        }
      }
    }

    int GetProcessorState() {return ProcessorState;}

    String LableString () {
      return (new String
  (Integer.toString(ProcessorNumber)+"/"+Integer.toString(taskNumber)));
    }

    void SetProcessorClockTime(double time) {clockTime = time;}

    double GetProcessorNextEventTime() {return clockTime;}

    void SetProcessorDurations(int task, double inputTime, double processTime, double
  outputTime){
      taskNumber = task;
      inputTimeDuration = inputTime;
      processTimeDuration = processTime;
      outputTimeDuration = outputTime;
      }

    void SetProcessorState(int state){
      ProcessorState = state;
      this.setBackground(StateColor());

      try {Thread.sleep((long) (0*1000.));} catch (InterruptedException e){;}
      this.repaint();
    }

    String CurrentState(){
    switch (ProcessorState){
    case 0:
    return new String("awaiting data");
    case 1:
    return new String("receiving data");
    case 2:
    return new String("processing");
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  case 3:
  return new String("waiting to output");
  case 4:
  return new String ("outputing data");
  default:
  return new String ("UNKNOWN STATE");
  }
  }
  Color StateColor(){
    int r = 127;
    int g = 127;
    int b = 127;
    if (ProcessorState == AwaitingData){
      r =   0; g = 128; b =   0;
    } else
      if (ProcessorState == RecvingInput){
        r = 255; g = 255; b = 102;
      } else
        if (ProcessorState == Processing){
          r =   0; g = 255; b =   0;
        } else
          if (ProcessorState == WaitingOutput){
            r =  51; g = 255; b = 255;
          } else
            if (ProcessorState == OutputingData){
              r =   0; g =   0; b = 255;
              }
    return new Color(r,g,b);
  }
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX E. COMPLETE RAY PROGRAM CONTROL FILE

This is a listing of the control file read by the Beowulf master program and then parsed into separate files for the slave processes. The slave processes read their tailored file prior to beginning execution.

```
input prof_file = "test1";
input bath_file = "test1";

output ascii_file = "ascii";

source range = 0 (m);
source depth = 10 (m);

receiver range = 10 (Nmi);
receiver depth = 100 (m);

angles first = 10 (degrees);
angles last = -11 (degrees);
angles number = 21;

fan dr = 375 (m);
fan min_range = 123 (m);
fan max_range = 2345 (Nmi);

add_extrema min_angle  = 123 (degrees);
add_extrema iterations = 4;
add_extrema min_dz     = 45 (m);
add_extrema z_miss     = 67 (m);

model   range_depend   =      full;
model   bottom_type     =     reflecting;
model   bath_smoothing  =        10 (km);
model   bottom_depth    =       6000 (m);
model   earth_radius   =      6378.137 (km);
model   z_tolerance    =      1e-06 (m);
model   margins        =      100;
model   max_bounces     =       900;
model   max_angle      =      100 (degrees);


eigenrays   geo_miss        =          5 (m);
eigenrays   search_max      =         10;
eigenrays   dr              =          0 (m);


step_size   multiplier      =          1;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
step_size   max            =        200 (m);
step_size   min            =          5 (m);
step_size   cos_factor     =         10;

prof_smoothing levitus;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX F. SOUND SPEED PROFILES AND THEIR USE IN ACOUSTIC RAY TRACING

The speed of sound in water has been under active investigation since 1827. At that time Collodon and Strurm measured the sound speed in Lake Geneva. In their experiment, an underwater bell was struck at the same time as a light was flashed and the difference in arrival time at an observer 13.5 km. away was recorded as 9.4 ± 0.2 seconds. Assuming that for these short distances that the speed of light was nearly instantaneous, the speed of sound was 13,500/9.4 = 1440 meters per second. The current measurement of sound speed under the same test conditions is 1439.2 m/s.

Since the days of Collodon and Strurm, many different ways have been developed to measure sound speed directly. Currently a velocimeter can be lowered from a boat and the speed of sound recorded as the velocimeter sinks into the water. When measurements are completed, the operator has a sound speed versus depth profile. As more and more sound speed profiles were collected, it was noticed that the speed of sound varies as a function of depth, temperature and salinity. An accurate sound speed profile combined with Snell's Law has been long recognized as the single most important item in predicting the path of an acoustic ray.

Fig. 39 shows nine years of sound speed data collected 15 miles SE of Bermuda. The sound speed profile shows four general areas of interest. They are:

- Diurnal layer: the section of the ocean from the surface to approximately 30 meters that is affected by daily heating and cooling,
- Seasonal thermocline: from the bottom of the diurnal layer to about 300 meters that is affected by seasonal changes in heating and cooling,
- Main thermocline: from the bottom of the seasonal thermocline to about 1000 meters that is only marginally affected by changes in the seasons,
- Deep isothermal layer: from the bottom of the main thermocline to the bottom of the ocean where the sound speed is affected primarily by water pressure (i.e., depth), and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The arrows show the minimum, normal and average sound speeds over the course of the measurement time.

The challenge was to find a closed form equation based on data that was readily available onboard ships that closely matched the measured data that could be used by the properly trained personnel.

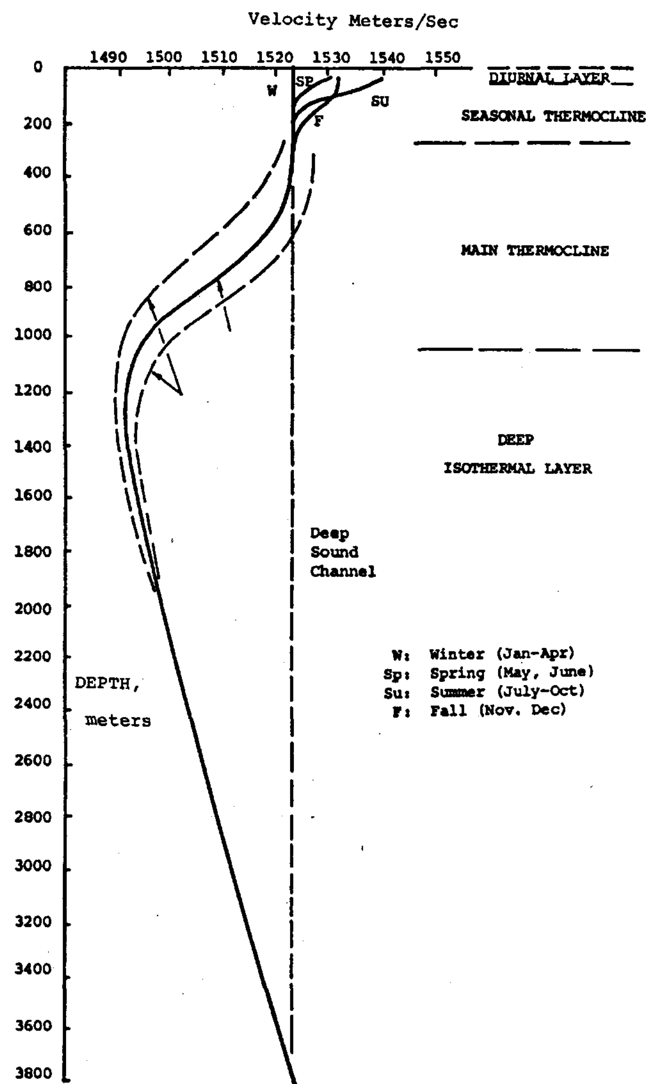


Fig. 39. Multi-year Sound Speed Profile From SE Of Bermuda

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

F.1 DERIVATION

During the 1940s, 1950s, and 1960s the US Navy expended a considerable amount of time and effort trying to find a single sound speed profile equation that fit most of the data collected by that time. Equations (25) through (28) are known respectively as Kuwahara's, Del Grosso's, Wilson's and Leroy's sound speed equations. All equations are still in use.

$$c = 1445.5 + 4.664T - 0.0554T^2 + 1.307*(S-35) + ... \tag{25}$$

$$c = 1448.6 + 4.618T - 0.0523T^2 + 1.25*(S-35) + ... \tag{26}$$

$$c = 1449.2 + 4.623T - 0.0546T^2 + 1.391*(S-35) + ... \tag{27}$$

$$\begin{aligned} c = {} & 1492.9 + 3*(T-10) - 6*10^{-3}*(T-10)^2 - 4*10^{-2}*(T-18)^2 \\ & + 1.2*(S-35) - 1*10^{-2}*(T-18)*(S-35) + Z/61 \end{aligned} \tag{28}$$

TABLE X shows the parameters, units and valid ranges of the components in the sound speed equations.

TABLE X. Valid Range Of Various Values For Sound Speed Profile Equations

	Symbol	Units	Lower Limit	Upper Limit
Temperature	T	Celsius	-2	24.5
Salinity	S	Practical salinity units	30	42
Depth	Z	Meters	0	1,000
Speed of sound	C	Meters per second	1433.04	1557.78

Additionally, there are other sound speed equations besides those listed here. Leroy's Equation (Eq. (28)) has the most wide spread use in current applications because it is relatively easy to program into digital

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

computers. Examination of Leroy's Equation shows that eventually the depth of the water (due to the pressure at depth) will dominate all other terms.

F.2 BATHYTHERMOGRAPH DATA

The use of Leroy's Equation in a shipboard environment is relatively easy. Most of the earth's oceans have a salinity of about 35 parts per thousand, so it can almost be treated as a constant, only varying under unusual circumstances. The temperature of the water can be measured via the use of a reusable bathythermograph or an expendable (XBT) probe. A handheld XBT launcher is shown in Fig. 40 and a cutaway XBT is shown in Fig. 41. Of particular interest in the cutaway are the two spools of wire, one in the probe and the other in the canister. The wire is made of two copper strands thinner than a human hair and therefore would not withstand any tension. When the XBT is launched/dropped, both spools start to unwind. The spool on the probe unwinds to compensate for the sinking of the probe, while the upper spool unwinds to compensate for the movement of the ship. The net effect of these unwindings is zero tension on the wire. The XBT sinks at a constant rate, so timed temperature measurements from the thermistor in the nose of the probe equate to the temperature at an assumed depth. These data can be fed directly into a computer. A complete system showing different types of XBT launchers is shown in Fig. 42. Armed with the temperature at depth, depth and salinity, shipboard personnel can use Leroy's Equation to compute a sound speed profile.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

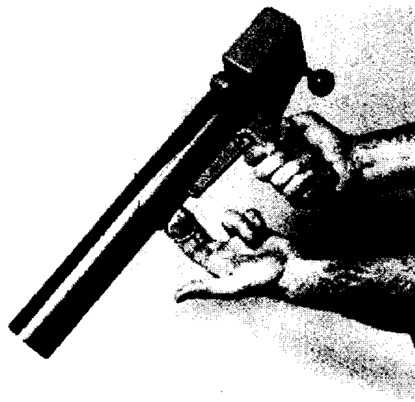


Fig. 40. Handheld XBT Launcher

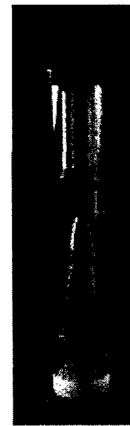


Fig. 41. Cutaway Of An XBT In Its Launch Barrel

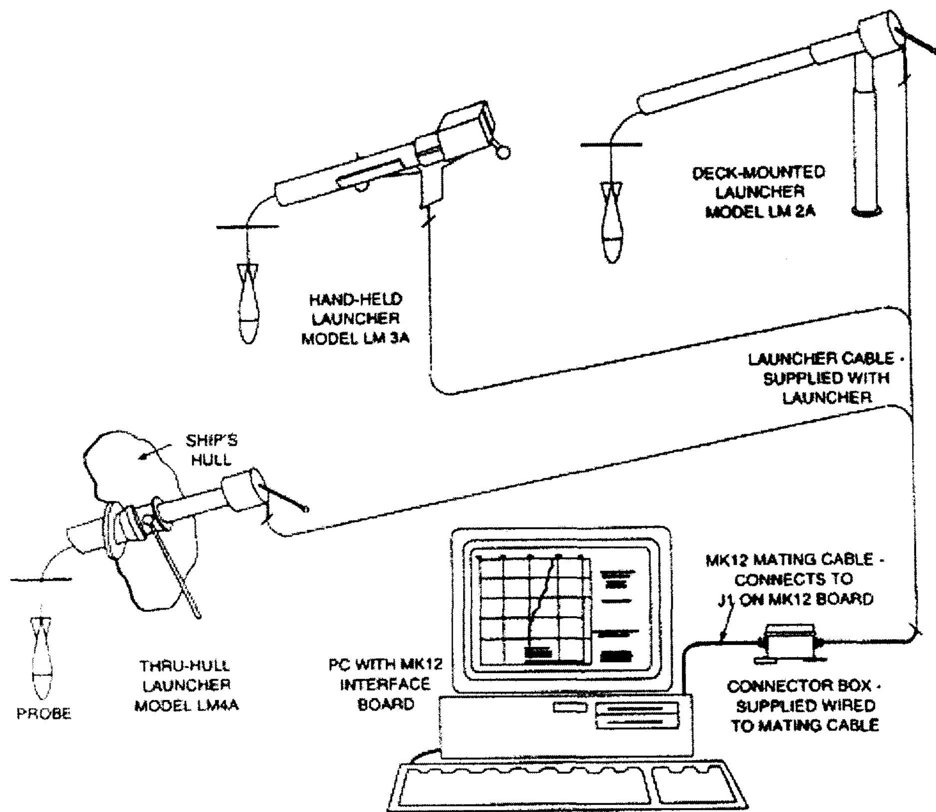


Fig. 42. Complete XBT Measurement System

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX G. ACOUSTIC RAY TRACING

G.1 THEORY

Snell's Law of Sines was derived in the field of optics. But its underlying premise that a ray will bend based on the relative speeds of the ray in a medium is fundamental to the study of acoustic rays in water. In the acoustic world, it is the speed of sound rather than light that is of importance and the speed of sound can vary within the same body of water. Appendix F contains a discussion on sound speed profiles, including their history, their derivation, how they are obtained and their importance.

Snell's Law is used to compute the path that one acoustic ray will travel away from the source (the sonar) and returning. The path is based on the relative speeds of sound in the water that the ray travels through based on the sound speed profile.

G.2 PRACTICE

A sonar is placed at a known depth in the sound speed profile and rays are "shot" from the sonar at known start angles. These individual angles correspond to the α angle in Snell's diagram, Fig. 30. The start angle can be conceived of as equivalent to the slope of a straight line in a Cartesian coordinate system. When the ray travels far enough in range (X axis) to intersect with the change in sound speed (Y axis) from the sound speed profile, β is computed. Now that the ray is in this new speed channel, the complement of β becomes the α angle for the next iteration. This procedure is repeated until the ray is far enough away from the sonar to be of little interest. Because the ray is "bending" (i.e., the α and β angles are changing) as it travels. The ray might impact the surface or the bottom.

G.2.1 Surface Reflection and Scattering

If the surface of the sea were perfectly smooth, it would make an almost perfect reflector of sound. As shown in Eq. (29), the criterion for the roughness or smoothness of the sea is the *Rayleigh* parameter.

$$R = \frac{2 * \pi}{\lambda} * H * \sin(\theta) \tag{29}$$

Where H is the rms "wave height" (crest to trough) and θ is the grazing angle. When $R \ll 1$, the surface is primarily a reflector. When $R \gg 1$, the surface is primarily a scatterer. An assumed wind speed, or a sea

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

state number can be used to compute λ (the length of the waves on the surface) or H (the height of the waves).

G.2.2 Bottom Reflection, Absorption and Scattering

The sea bottom is a reflecting, absorbing and scattering medium that is much more complex than the surface. There are two main reasons that the seabed is vastly more complex. The first is the composition of the seabed that may vary from hard rock to soft mud. Secondly, it is often layered, with a density and a sound speed that can change gradually or abruptly with depth within the bottom. For these reasons, the reflection, absorption and scattering of the bottom is less easily predicted than the surface. Most ray trace programs use a set of predefined values to compute bottom loss as a function of bottom type (sand, mud, bottom, etc.) and sonar frequency. A complete discussion of sonar losses associated with various bottoms is beyond the scope of this paper.

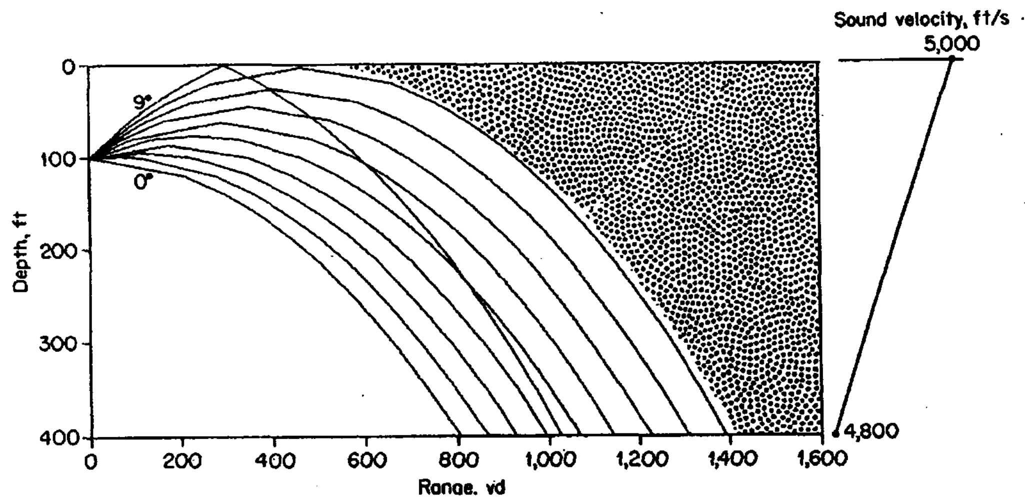


Fig. 43. Surface Shadow Zone

Fig. 43 and Fig. 44 show the effects of different linear gradient sound speed profiles on rays. In Figure 42 a constantly decreasing speed will cause rays to bend downward with the result that there are no rays in the stippled area. Rays will be bent back up eventually as the profile is affected by depth. The distance between the range where the rays are initially bent downward and when they return is called a surface shadow zone. The amount of acoustic energy in the shadow zone can be 40 to 60 dB below the non-shadow zone.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

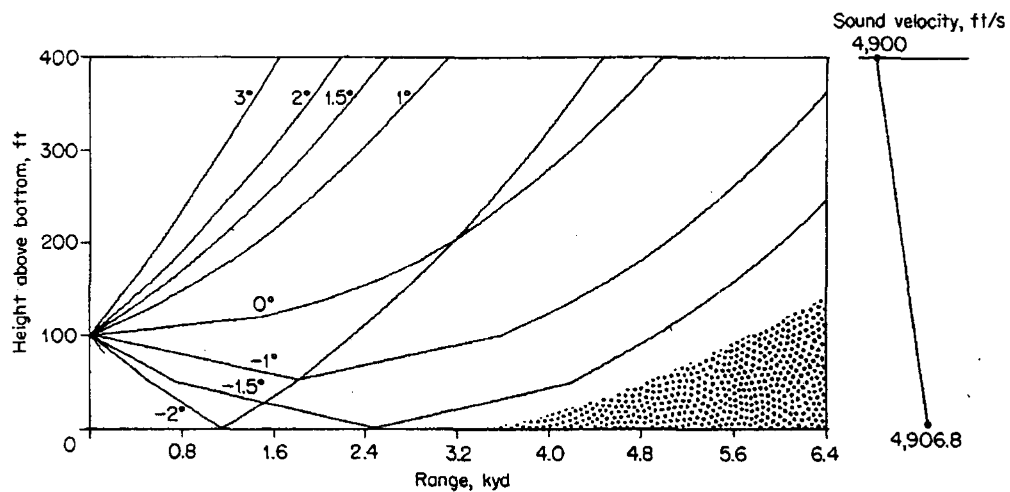


Fig. 44. Bottom Shadow Zone

In Fig. 44 the rays are again bent towards the slower speed. This time, it results in a shadow zone near the bottom until the rays are refracted/reflected down.

The interaction of the rays is based on repeated application of Snell's Law and can result in very complex ray paths. A ray trace program computes where a ray should go as well as computing the length of the ray (as differentiated from the distance the ray is from the sonar) and the number of surface and bottom bounces that may have occurred along the ray's path. These data are significant factors in computing the amount of energy that is present at the sonar's acoustic wave front relative to energy existed between two rays that started adjacent to each other at the sonar's face.

G.3 SOUND SPEED CHANNELS

Snell's Law has an interesting behavior based on the origin of the ray and the ray's intersection with the next speed channel. A ray originating an angle α in a lower index area intersecting at a higher index area bending to angle β has already been explored. .

As shown in Fig. 45, a ray originating in the higher index area going towards the lower has two ways that it can behave.

Equations (30) and (31) are restatements and reordering of Snell's Law of Sines. By definition and shown

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

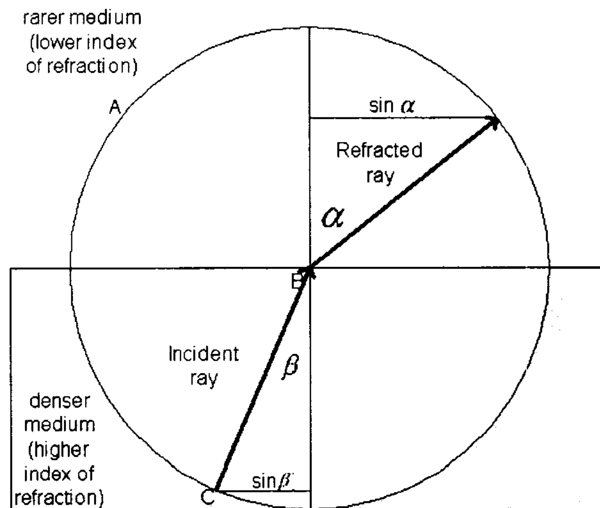


Fig. 45. Snell Ray Originating in Denser Medium

$$\frac{\sin(\alpha)}{\sin(\beta)} = \frac{H}{L} \quad (30)$$

$$\sin(\alpha) = \sin(\beta) * \frac{H}{L} \quad (31)$$

$$0 \leq \sin(\alpha) \leq 1 \quad (32)$$

$$1 \leq L < H \quad (33)$$

$$\sin(\alpha) = 0 \Rightarrow \sin(\beta) = 0 \quad (34)$$

$$\sin(\alpha) = 1 \Rightarrow \sin(\beta) = \frac{L}{H} \quad (35)$$

$$\theta = \sin^{-1}(\frac{L}{H}) \quad (36)$$

in (32), sine α is bounded between 0 and 1. Equation (33) states that the indices of refraction are bound by the relationship that the lower index is greater than or equal to 1 and the higher index is greater than the lower. β is relative to normal to the boundary between the two mediums, thus β has to be between 0 and 90 degrees. By (34), the sine of α will be 0 when the sine β is 0, or the higher index is 0. A higher index equaling 0 contradicts the basic boundaries on the indices and so is impossible. Sine α equaling 1, in Eq.

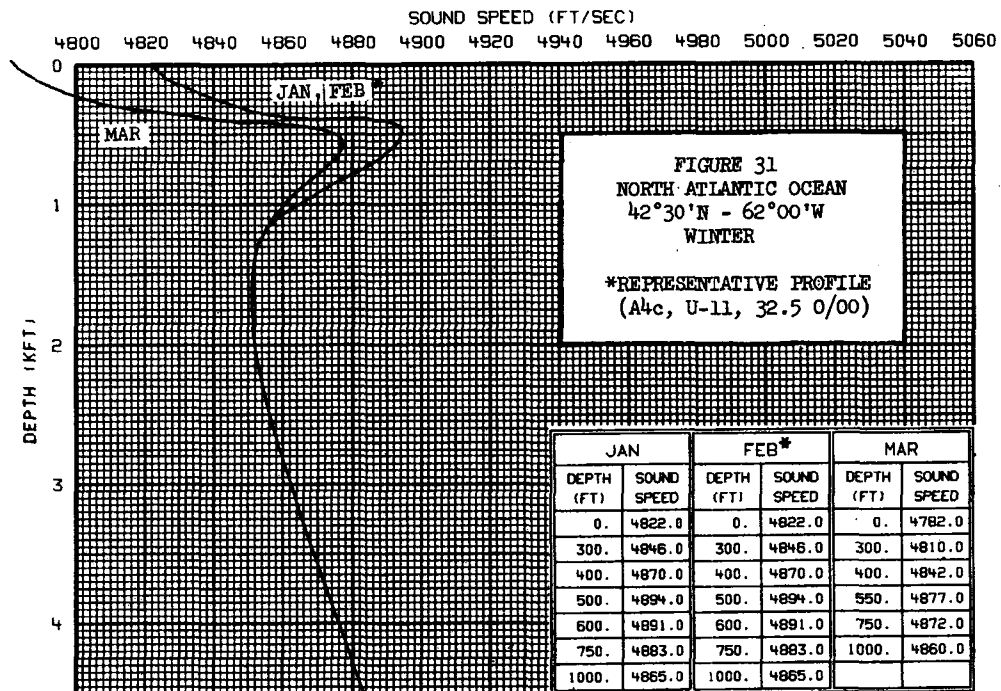
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(35) is possible if sine β equals the ratio of the lower index to the higher index. The ratio of the lower to the higher index defines a limit, known as the critical angle limit for the β, as shown in Eq. (36). Any value of β less than or equal to the critical angle will result in a correct value for α. Angle β greater than the critical angle will cause the ray to be reflected back into the high index channel at an angle equal to the angle of incidence. The critical angle for a water to air interfaces is 48.6 degrees.

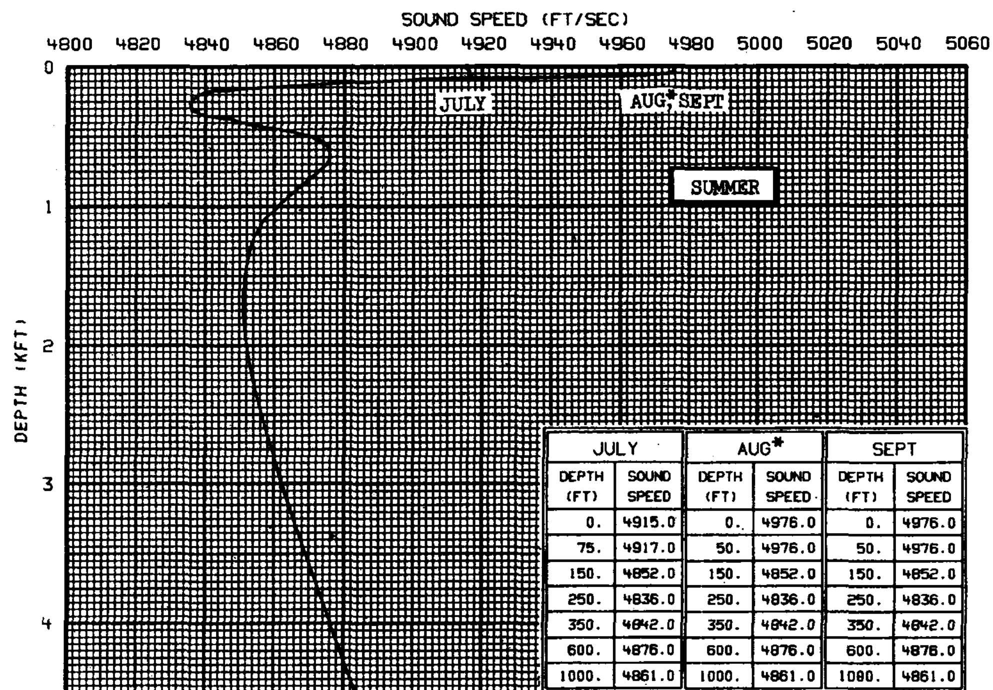
Fig. 46 shows two sound speed profiles from the area approximately 180 Nm south of Nova Scotia, Canada with significant sound channels [12]. A ray that is started in the summer upper sound channel (approximately 300 feet from the surface) may be trapped forever in the channel depending on the angle that the ray intersects the boundary at approximately 600 feet. A different ray in the deep sound channel will be trapped between 600 feet and the bottom. For a ray to remain trapped, it must intercept the changes in the sound speed profile at an angle less then the critical angle.

Snell's Law, Eq. (23) is also at the heart of the way modern fiber optic cables operate [15]. A light ray is "shot" down the fiber optic cable at an angle greater than the Snell's critical angle (36). Snell's critical angle is computed based on the relative indices of the mediums, in the case of fiber optic cable, one medium is the center conductor and the other is the outer sheathing. Light "shot" at an angle greater than the critical angle, will not escape from the medium with the higher index and will remain trapped in the medium.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



JAN		FEB*		MAR	
DEPTH (FT)	SOUND SPEED	DEPTH (FT)	SOUND SPEED	DEPTH (FT)	SOUND SPEED
0.	4822.0	0.	4822.0	0.	4782.0
300.	4846.0	300.	4846.0	300.	4810.0
400.	4870.0	400.	4870.0	400.	4842.0
500.	4894.0	500.	4894.0	550.	4877.0
600.	4891.0	600.	4891.0	750.	4872.0
750.	4883.0	750.	4883.0	1000.	4860.0
1000.	4865.0	1000.	4865.0		



JULY		AUG*		SEPT	
DEPTH (FT)	SOUND SPEED	DEPTH (FT)	SOUND SPEED	DEPTH (FT)	SOUND SPEED
0.	4915.0	0.	4976.0	0.	4976.0
75.	4917.0	50.	4976.0	50.	4976.0
150.	4852.0	150.	4852.0	150.	4852.0
250.	4836.0	250.	4836.0	250.	4836.0
350.	4842.0	350.	4842.0	350.	4842.0
600.	4876.0	600.	4876.0	600.	4876.0
1000.	4861.0	1000.	4861.0	1080.	4861.0

Fig. 46. Sound Speed Profile Showing Two Sound Channels

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX H. ROBUST LINEAR LEAST SQUARES

During the data analysis phase of the investigation, it appeared that there was a linear relationship between the number of processors and the execution time for a given number of rays. This relationship was first apparent by visual inspection of the data. A linear relationship is defined as Eq. (37).

$$y = mx + b \tag{37}$$

The computation of m and b as shown in Eq. (37) can be found in many introductory statistics books [8] and is often included as part of introductory computer programming courses.

One of the underlying assumptions in the computation of m and b is that any errors in the data form a Gaussian distribution on each data point. Standard least squares curve fitting assumes that all data is valid, thereby treating each data point equally. This assumption causes any data points that are far outside the Gaussian distribution to have an unwarranted effect on the solution.

Robust least squares makes no assumptions about the error distribution in the data. An estimate of the coefficients is computed and than applied to all data points. The distance between the real data point and the computed data point is used to "weigh" the real data. Points that are far away from the computed line, are weighed less than points that are close to the line. Robust least squares continues to work on a data set until the differences between consecutive iterations falls below some pre-set threshold, or the solution fails to converge after a pre-set number of iterations.

Visual inspection of the data clearly reveals that the (140,1.78) data point is grossly out of bounds. Standard least squares treated the data point as valid, while robust least squares arrived at a solution that more closely matched the majority of the data. Eq. (38) is the standard least squares fit to the data. Eq. (39) is the robust least squares fit.

TABLE XI [8] has a series of data points that are plotted in Fig. 47 along with standard and robust least squares fits.

Visual inspection of the data clearly reveals that the (140,1.78) data point is grossly out of bounds. Standard least squares treated the data point as valid, while robust least squares arrived at a solution that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

more closely matched the majority of the data. Eq. (38) is the standard least squares fit to the data. Eq. (39) is the robust least squares fit.

TABLE XI. Valid Range Of Various Values To Demonstrate Least Squares Curve Fitting

Air velocity (cm/sec)	Evaporation coefficient (mm^2/sec)	
20	0.18	
60	0.37	
100	0.35	
140	1.78	<= Changed from the original of 0.78 for illustration
180	0.56	
220	0.75	
260	1.18	
300	1.36	
340	1.17	
380	1.65	

$$StdardLeastSquares = 0.0037400*x + 0.2601250 \quad (38)$$

$$RobustLeastSquares = 0.0039284*x + 0.032286 \quad (39)$$

The robust least squares fit arrived at its solution after 5 iterations.

The following pages contain the robust least squares program and make file that was used to analyze the Beowulf performance data. The robust least squares program is based on newmat10A. Data for Fig. 47 was produced using the command: ./testing 0 6.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

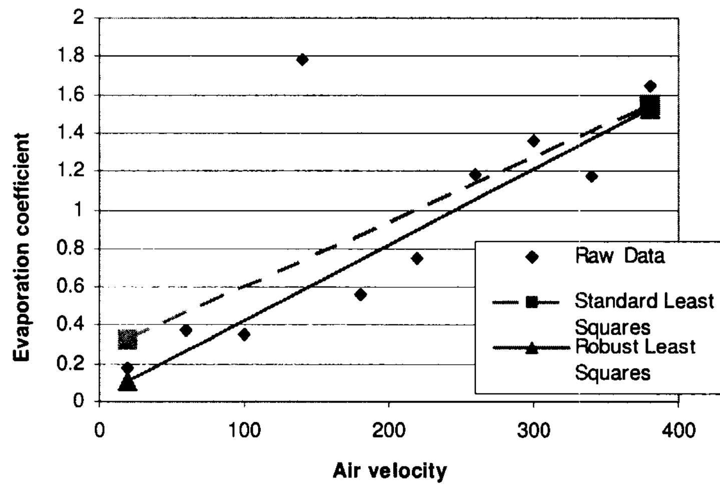


Fig. 47. Comparison of Standard and Robust Least Squares Curve Fitting

H.1 THE MAKE FILE:

```
CXX = g++

LIBRARIES=-lnewmat \
            -lm

LIBDIR=-L../Lib

INCLUDES=-I ../Include

CXXFLAGS = -O2 $(LIBDIR) $(LIBRARIES) $(INCLUDES)

all:testing

testing:../Src/testing.cc
            $(CXX) -o ../testing ../Src/testing.cc $(CXXFLAGS)

run:
            ../testing

clean:
            rm -f ../testing
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

H.2 THE SOURCE FILE TESTING.CC:

```
#include <iostream>
#include <list>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "newmat.h"

#define DEBUG_PRINT

#ifdef DEBUG_PRINT
#define DEBUG_PRINT(...) {fprintf (stderr,"%s() in %s at %d: ",__FUNCTION__, __FILE__, __LINE__);fprintf
(stderr,__VA_ARGS__);};
#define DEBUG_VARIABLE(v,fmt) {DEBUG_PRINT(#v " =>%"#fmt "<=\n",v);};
#else
#define DEBUG_PRINT(...)
#define DEBUG_VARIABLE(...)
#endif

using namespace std;

void matrixFunction (ColumnVector *x, double (*func) (double value)){
 for (int i = 1; i <= x->Nrows(); i++)
  x->element(i-1) = ((*func) ((double) x->element(i-1)));
}

void matrixDump(char title[], char matrixName[], Matrix x){
 DEBUG_PRINT("%s\n",title);
 for (int i= 1; i <=x.Nrows(); i++){
  for (int j = 1; j<=x.Ncols(); j++)
   DEBUG_PRINT("%s [%d][%d] = %f\n", matrixName, i, j, x(i,j));
 }
}

void leastSquares (ColumnVector x, ColumnVector y, ColumnVector *yHat, ColumnVector *r, ColumnVector *b, Matrix w){

 Matrix X (x.Nrows(),2);

 for (int i = 1; i<= x.Nrows(); i++){
  X(i,1) = x(i);
  X(i,2) = 1;
 }

 Matrix B = (X.t() * w * X).i() * (X.t() * w * y);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  matrixDump("In leastSquares B","B",B);
  Matrix YHat = (X * B);
  // matrixDump ("Least squares","YHat",YHat);
  // Copy internal data to the outside
  for (int i = 1; i <= YHat.Nrows(); i++){
    yHat->element(i-1) = YHat(i,1);
    r->element(i-1) = (y(i) - yHat->element(i-1));
  }

  b->element(0) = B(1,1); // The slope of the line.
  b->element(1) = B(2,1); // The y-intercept
}

double matrixMedian(ColumnVector r){
  list <double> values;
  int size;
  double returnValue;

  for (int i = 1; i <= r.Nrows(); i++){
    values.push_back(r(i));
  }

  values.sort();
  size = values.size();

  if (size & 1){ // an odd number of entries
    for (int i = 1; i <= (int)(size/2); i++)
      values.pop_front();
    returnValue = values.front();
  }else{ // an even number of entries
    for (int i = 1; i <= (size/2); i++){
      returnValue = values.front();
      values.pop_front();
    }
    returnValue += values.front();
    returnValue /= 2.;
  }
  return returnValue;
}

int robustLeastSquares (ColumnVector x, ColumnVector y, ColumnVector *yHatOut, ColumnVector *rOut, ColumnVector *bOut){
  // Based on DataPlot manual ,Weights section
  ColumnVector delta(y.Nrows()),  delta2(y.Nrows());
  ColumnVector r(y.Nrows()), rOld(y.Nrows());
  ColumnVector temp(y.Nrows());
  ColumnVector u(y.Nrows()), tag(y.Nrows());
  ColumnVector yHat(y.Nrows()), b(2);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
ColumnVector yHatOld(y.Nrows()), bOld(2);

double c = 4.685;
double denom;
double epsilon = 0.0001;
double mad;
double median;
double num;
double s;

int iterationCounter;
int maxIterations = 10;
int nanFlag = 0;

matrixDump("Original x values","x", x);
matrixDump("Original y values","y", y);

Matrix w (x.Nrows(),x.Nrows());

w = 0.0;
for (int i = 1; i <=x.Nrows(); i ++)
  w(i,i) = 1;

// Initialize the residual vector
leastSquares (x, y, &yHat, &r, &b, w);

yHatOld = yHat;
bOld = b;

for (iterationCounter = 1;
    iterationCounter < maxIterations;
    iterationCounter++){
  rOld = r;
  median = matrixMedian(r);

  temp = r - median;
  // Apply the fabs function to all elements of the temp matrix
  matrixFunction(&temp, fabs);
  //   matrixDump ("robustLeastSquares","temp",temp);
  mad = matrixMedian(temp);

  s = mad/0.6745;
  u = r / s;
  tag = u / c;
  //   matrixDump ("robustLeastSquares tag","tag",tag);

  matrixFunction(&tag, fabs);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for (int i = 1; i <= x.Nrows(); i++){
 if(tag(i) > 1)
        w(i,i) = 0;
 else
        {
         double localDouble = tag(i);
         localDouble *= localDouble;
         localDouble = 1 - localDouble;
         w(i,i) = localDouble * localDouble;
        }
}
// Compute a weighted least squares fit with new weights
leastSquares (x, y, &yHat, &r, &b, w);

delta = (rOld - r);
for (int i = 1; i <= x.Nrows(); i++)
 delta(i) = delta(i) * delta(i);

num = delta.Sum();
DEBUG_VARIABLE(num,f);
if (isnan(num) != 0){
 nanFlag = 1;
 break;
}
num = sqrt(num);

for (int i = 1; i <= x.Nrows(); i++)
 delta2(i) = rOld(i) * rOld(i);

denom = delta2.Sum();
DEBUG_VARIABLE(denom,f);
if (isnan(denom) != 0){
 nanFlag = 1;
 break;
}
denom = sqrt (denom);
/*
 If the difference between the last iteration and this
 is less then the limit then we are finished
*/
if (denom == 0) { // Houston we have closure!
 DEBUG_PRINT("denom == 0, we have closure!!\n");
 break;}
DEBUG_PRINT("iteration %d closeness %f\n", iterationCounter, (num/denom));
if ((num/denom) < epsilon)
 break;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  // Save the old values
  yHatOld = yHat;
  bOld = b;
 }

 if (nanFlag){ // Something is a nan, so use old data
  for (int i = 1; i <= yHat.Nrows(); i++){
   yHat(i) = yHatOld(i);
  }

  b(1) = bOld(1);
  b(2) = bOld(2);
 }

 // Copy internal data to the output matrices

 for (int i = 1; i <= yHat.Nrows(); i++){
  yHatOut->element(i-1) = yHat(i);
  rOut->element(i-1) = y(i) - yHat(i);
 }

 bOut->element(0) = b(1); // Also known as the slope of the line
 bOut->element(1) = b(2); // The y-intercept.

 return (iterationCounter);
}

void matrixTester(ColumnVector x, ColumnVector y){
 Matrix w (x.Nrows(),x.Nrows());
 ColumnVector yHat(y.Nrows()), r(y.Nrows()),  b(2);

 w = 0.0;
 for (int i = 1; i <= x.Nrows(); i++)
  w(i,i) = 1.0;

 leastSquares (x, y, &yHat, &r, &b, w);

 matrixDump("Tester funtion b (simple least squares function)","b", b);
 //  matrixDump("Function r","r",r);
 //  matrixDump("Function yHat","yHat",yHat);
 cout << "robustLeastSquares iterations = " << robustLeastSquares (x, y, &yHat, &r, &b) << endl;
 cout << "y = " << b.element(1) << " + "<< b.element(0) << "*x"  << endl;
 matrixDump("Funtion robustLeastSquares b","b", b);
 //  matrixDump("Function robustLeastSquares yHat","yHat",yHat);
}

void matrixTestMedian(char testName[], ColumnVector x, double expectedValue){
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  double matrixReturn = matrixMedian(x);
  cout << "matrixMedian() ";
  if (matrixReturn == expectedValue)
    cout << "passed";
else
  cout << "failed";

  cout << " " << testName << " test, returning " << matrixReturn << " when expecting " << expectedValue << endl;
}

int main(int argc, char *argv[]){
  ColumnVector x;
  ColumnVector y;
  int categoryTest;
  int subCategoryTest;
  int returnValue = 0;

  categoryTest = 0;
  subCategoryTest = 5;

  if (argc>1){
    switch (argc){
    case 3:
      subCategoryTest = atoi(argv[2]);
    case 2:
      categoryTest = atoi(argv[1]);
      break;
    default:
      printf ("Usage: %s [categoryTest [subCategoryTest]]\n exit(-99)\n",argv[0]);
      exit(-99);
    }
  }

  switch (categoryTest){
  case 0: // Data from "good" soruces
    switch (subCategoryTest){
    case 0: // Data from Probability and Statistics for Engineers
      x.ReSize (10);
      y.ReSize (x.Nrows());
      x << 20 << 60 << 100 << 140 << 180 << 220 << 260 << 300 << 340 << 380;
      y << 0.18 << 0.37 << 0.35 << 0.78 << 0.56 << 0.75 << 1.18 << 1.36 << 1.17 << 1.65;
      break;
    case 1: // Data from DataPlot Manual, Weights section
      x.ReSize (10);
      y.ReSize (x.Nrows());
      x << 1 << 2 << 3 << 4 << 5 << 6 << 7 << 8 << 9 << 10;
      y << 2 << 4 << 60 << 7 << 9 << 12 << 14 << 15 << 18 << 20;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 break;
case 2:// Data from http://www.cs.bsu.edu/homepages/fischer/math125/regression.pdf
 x.ReSize (4);
 y.ReSize (x.Nrows());
 x << 2 << 6 << 8 << 4;
 y << 5 << 4 << 9 << 2;
 break;
case 3:// Data from http://www.udayton.edu/~cps/csp353/lsqd/lsqd.html
 x.ReSize (10);
 y.ReSize (x.Nrows());
 x << 1 << 2 << 3 << 4 << 5 << 6 << 7 << 8 << 9 << 10;
 y << 1.3 << 3.5 << 4.2 << 5 << 7 << 8.8 << 10.1 << 12.5 << 13 << 15.6;
 break;
case 4: // Data from my mind
 x.ReSize (30);
 y.ReSize (x.Nrows());

 x(1) = 0;
 x(2) = 1;
 for (int i = 3; i <= x.Nrows(); i++)
        x(i) = x(i-1) + x(i-2); // Fibernaci series

 for (int i = 1; i <= x.Nrows(); i++){
        y(i) = (0.23456 * x(i) + 8.9) + 30 *( (double)rand()/(double)RAND_MAX);
        if (i & 4)
         y(i) += 100 *( (double)rand()/(double)RAND_MAX);
 }
 break;
case 5: // Data from Probability and Statistics for Engineers
 x.ReSize(5);
 x << 15 << 14 << 2 << 27 << 13;
 matrixTestMedian ("first",x,14);
 x.ReSize(6);
 x << 17 << 9 << 15 << 19 << 4 << 16;
 matrixTestMedian ("second",x,15.5);
 matrixTestMedian ("second (intentional failure)",x,15);
 returnValue = -4;
 break;
case 6: // Data from Probability and Statistics for Engineers
 // with some GROSSLY wrong
 x.ReSize (10);
 y.ReSize (x.Nrows());
 x << 20 << 60 << 100 << 140 << 180 << 220 << 260 << 300 << 340 << 380;
 y << 0.18 << 0.37 << 0.35 << 1.78 << 0.56 << 0.75 << 1.18 << 1.36 << 1.17 << 1.65;
 break;
default:
 returnValue = -1;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  };
  break;
case 1: // Data from real time Beowulf paper
  x.ReSize (7);
  x << 1 << 20 << 20 << 30 << 100 << 200 << 1000;

  y.ReSize (x.Nrows());
  switch (subCategoryTest){
  case 1: // Serial execution times
    y << 0.0120 << 0.0560 << 0.1020 << 0.1510 << 0.4860 << 0.9680 << 4.4820;
    break;
  case 3: // 3 slaves execution times
    y << 0.5690 << 0.7220 << 0.8920 << 1.0580 << 2.3280 << 4.0790 << 17.3910;
    break;
  case 4: // 4 slaves execution times
    y << 0.5730 << 0.7110 << 0.9030 << 1.0280 << 2.2600 << 3.8950 << 17.6340;
    break;
  case 5: // 5 slaves execution times
    y << 0.5760 << 0.7160 << 0.8700 << 1.0360 << 2.1830 << 3.9070 << 17.2480;
    break;
  case 6: // 6 slaves exectuion times
    y << 0.5780 << 0.7290 << 0.8740 << 1.0330 << 2.1210 << 3.7510 << 17.3600;
    break;
  case 7: // 7 slaves execution times
    y << 0.5860 << 0.7190 << 0.8750 << 1.0300 << 2.1860 << 3.84720 << 17.0570;
    break;
  default:
    returnValue = -2;
  }
  break;
default:
  returnValue = -3;
}

switch (returnValue){
case 0:{
  // Where the work starts

  matrixDump("Original x values","x", x);
  matrixDump("Original y values","y", y);

  Matrix X (x.Nrows(),2);

  for(int i = 1; i <=x.Nrows(); i++){
    X(i,1) = x(i);
    X(i,2) = 1;
  }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// Least squares as a series of explicit functions

Matrix b = (X.t() * X).i()* (X.t() * y);
Matrix yHat = (X * b);
Matrix r = (y - yHat);
//   matrixDump("Explicit b","b", b);

// matrixDump("Computed y values","y", yHat);
// matrixDump("Resdiuals","r", r);

// Weighted least squares as a series of explicit calls
Matrix W (X.Nrows(),X.Nrows());
W = 0.0;

// matrixDump("Weights before computation","W", W);
for (int i = 1; i <= W.Nrows(); i ++)
  W(i,i)= 1/(r(i,1)*r(i,1));

// matrixDump("Weights","W", W);

Matrix b2 = (X.t() * W * X).i() * (X.t() * W * y);
Matrix yHat2 = (X * b2);
Matrix r2 = (y - yHat2);

// matrixDump("Weighted b","b2", b2);
// matrixDump("Weighted y","yHat2", yHat2);
// matrixDump("Weighted Resdiuals","r2", r2);

Matrix bDiff = (b - b2);

// matrixDump("b differences","bDiff", bDiff);
matrixTester(x,y);
}
break;
case -3:
case -2:
case -1:
 cout << "No action defined where categoryTest = " << categoryTest;
 cout << " and subCategoryTest = " << subCategoryTest;
 cout << endl;
 cout << "exit (" << returnValue <<")" << endl;
 exit(returnValue);
 break;
case -4: // Everything OK, just return from main()
 returnValue = 0;
 break;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 default:
  cout << "No action defined where returnValue = " << returnValue;
  cout << ", main() returning." << endl;
 }
 return (returnValue);
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

Charles Lane Cartledge was born in Los Angeles, California on 2 March 1952. He was raised in central Alaska, graduating from high school in Fairbanks, Alaska in 1970, the University of Alaska with an AEET in 1972, Oregon Institute of Technology with a BEET in 1974 and Old Dominion University with a MS in CS in 2007. He has worked at EDO Corporation Combat Systems in a variety of roles since joining them in 1983. He now serves in the dual role of Computer Systems Engineer and Principal Software Engineer with oversight of combat system design. Focusing on state-of-the-art defense technology for the Department of Defense, United States and foreign allied nations, Mr. Cartledge's position principally involves design of sonars, combat and data link systems. He owns a proven record of delivering results through the use of technology and creative problem solving, as well as meeting challenges head on. Overseeing project management, timelines, schedules and assignments; calculating and managing the engineering budget; providing assistance and advice to middle and upper management in addition to internal and external customers. Mr. Cartledge attributes his success to his supportive wife, Mary, their son, Charles Lane and to the time that he spent on active duty in the United States Navy. Mentoring and identifying entry level personnel's potential; directing design and deployment of high-level defense projects; providing quality training to others; and helping EDO solve critical system problems are some of his career highlights. He is a CAPT in the United States Navy Reserves, a Grand Croix in the Sovereign Military Order of the Temple Of Jerusalem Knight Templar, a former Boy Scouts of America Scout Master, and a former board member of Vets House (a transitional house for US Veterans in need).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.