1998

Parallelization and Implementation of Approximate Root Isolation for Nonlinear System by Monte Carlo.

Ebrahim Khosravi
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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 bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI 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.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

PARALLELIZATION AND IMPLEMENTATION OF APPROXIMATE ROOT ISOLATION FOR NONLINEAR SYSTEM BY MONTE-CARLO

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Ebrahim Khosravi
B.S. , Southern University, 1985
M.S. , Southern University, 1993
May 1998

DEDICATION

This work is dedicated to my mother, Agdas Oskoiezadeh, who always expected the best in me and to my wife Parrisa Khosravi, who always motivated me to be my best.

ACKNOWLEDGEMENTS

In the name of God, the beneficent, the merciful. I have been blessed to have the opportunity to be able pursue my education. I would like to express my appreciation to Dr. J. Bush Jones for his support, understanding and encouragement. His expertise was an invaluable asset to me and instrumental part of the success and completion of this research.

Sincere and deep gratitude extended Dr. S. S. Iyengar, Dr. Doris Carver, and Dr. Bob. R. Svoboda for their assistance, unselfish support and leadership which it helped me to deal with adversity and inspired me to excel. I would Like to offer a special thanks to my committee members whose supervision of my research inspired me to excellence. My special appreciation is expressed to those individuals who have helped me directly or indirectly during these years, where their names are not mentioned here.

I wish to gratefully acknowledge my parents for thier support and encouragement toward my eductation, especially my mother who always strived to provide me with the best. How do I find the words to express my appreciation to my family, especially my wife Parrisa Daraie for her patience and continuous support, encouragement, and good persion food during these years, I must say that your support made all of the difference in the world. At the end I would like thank our children, Seena, Roya, and Neema who have been the sources of joy and happiness during these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation solves a fundamental problem of isolating the real roots of nonlinear systems of equations by Monte-Carlo that were published by Bush Jones. This algorithm requires only function values and can be applied readily to complicated systems of transcendental functions. The implementation of this sequential algorithm provides scientists with the means to utilize function analysis in mathematics or other fields of science. The algorithm, however, is so computationally intensive that the system is limited to a very small set of variables, and this will make it unfeasible for large systems of equations. Also a computational technique was needed for investigating a metrology of preventing the algorithm structure from converging to the same root along different paths of computation. The research provides techniques for improve the efficiency and correctness of the algorithm.

The sequential algorithm for this technique was corrected and a parallel algorithm is presented. This parallel method has been formally analyzed and is compared with other known methods of root isolation. The effectiveness, efficiency, enhanced overall performance of the parallel processing of the program in comparison to sequential processing is discussed. The message passing model was used for this parallel processing, and it is presented and implemented on Intel/860 MIMD architecture.

The parallel processing proposed in this research has been implemented in an ongoing high energy physics experiment: this

algorithm has been used to track neutrinoes in a super K detector. This experiment is located in Japan, and data can be processed on-line or off-line locally or remotely.

CHAPTER 1

OVERVIEW OF ROOT ISOLATION

1.1 INTRODUCTION

The determination of the roots of nonlinear system of equations is a common and difficult problem in applied mathematics. Finding a single root of nonlinear equations or system of equations has often been a problem in the past. The new technique that was developed by Jones and Iyengar [Jones 94], shows promise for finely isolating or finding all roots of relatively large nonlinear systems over a given bounded region.

The general problem of solving the roots of nonlinear system of equations is to obtain disjoint intervals of approximately a given size which contain the real roots of a continuous function over a specified region. Commonly, when a root isolation strategy is restricted to use only function values, an interval is merely subdivided and searched for sign changes or graphed. To be certain of isolating the roots, one must make the subdivision fine; however, even with a fine subdivision, roots in which the function only touches but does not cross the axis are usually missed. Further, such searches are costly over large intervals. The idea developed will isolate roots even when the function does not cross the axis, and it efficiently searches large intervals. In the following section we will provide some background to the root finding problem, methods, and the solutions that are used.

1.2 BACKGROUND

One of most basic tasks in mathematics is solving equations. This section provides background about widely used nonlinear systems of equations and root finding. Multiple roots, or very close roots may exist in systems, especially if the multiplicity is an even number or some of the roots may be closely clustered [Cuyt 87].

Most equations have two parts: a right-hand side and a left-hand side. Traditionally, all terms are found on the left and solutions appear to the right.

$$f(x) = 0$$

When there is only one independent variable, the problem is said to be one-dimensional. Problems with more than one independent variable are said to be n-dimensional and have more than one equation to be satisfied simultaneously. Sometimes a nonlinear set of equations either may have no real solution at all or may have more than one solution. It is much more difficult to find roots for the simultaneous solution of equations in n- dimensions than in the one-dimensional case.

The principle difference between one and multi dimensions is that in one dimension, it is possible to trap a root between bracketing values, and then converge to it. In multi dimensions, one can never be sure that the root is there at all until the root is found. For example, to find the root of a one-dimensional problem, we start by getting some idea of what the function looks like. Next, we try to find its roots, and if this requires mass-produced roots for many different functions, then

we should at least know what some typical members of the ensemble look like.

Next, we should always bracket a root that is, know that the function changes sign in an identified interval before trying to converge to the root's value. The implicit function theorem states that generically the solutions will be distinct, point like, and separate from each other [Hove 70]. If, however, we should have a nongeneric (degenerate) case, then is possible to get a continuous family of solutions. For example, in vector notations, it is necessary to find one or more n-dimensional solution vectors (such as f(x)=0) in which f is the n-dimensional vector-valued function whose components are the individual equations to be satisfied simultaneously.

1.3 ROOT FINDING METHOD

In most cases the root finding invariably proceeds by iteration, and this is equally true in one or multidimensional equations. However, one cannot over emphasize the importance of a good first guess for the solution, especially for multidimensional problems. The table below shows the quick comparison of some well known method.

Table 1.1 Root finding methods comparison

METHODS	ADVANTAGES	DISADVANTAGES.
BISECTION	Simple	One root only
NEWTON	More then one roots	Good initial guess
SECANT	Smooth or linear function	May fail in multiplicity
GRAPHICAL	crude estimates	Not for large function

This crucial beginning usually depends with an analysis rather than numeric methods and carefully crafted initial estimates will help reduce computational effort. In the next sections of this chapter we will consider some well known root isolation methods. Detailed description of these methods are available in the bibliography.

1.3.1 BISECTION METHOD

The idea of bisection method is simple: over some interval, the function is known to pass through zero because it changes the sign. So the function is evaluated at mid point and its sign examined [Davi 88]. The midpoint is used to replace whichever limit has the same sign; after each iteration the root is known to be within an interval of size. After the next iteration, it will be bracketed within an interval of size.

The bisection method is very good method, but for only some functions. This method will work if the interval happens to contain one root. If the interval has more then one root or no root at all the Bisection method will fail.

1.3.2 SECANT METHOD

This method will work well with functions that are smooth near root, and it will converge faster then the bisection method. With the secant method, the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken at the point where the approximating line crosses the axis. After each iteration, one of the previous boundary points is discarded in favor of the latest estimate of the root. In general the secant method convergence of the algorithm may fail or become slow for roots of

multiplicity greater then one. In some cases this method may fail in multiplicity [Rals 78].

1.3.3 NEWTON METHOD

Consider the case of only two dimensions in which we want to solve function f and g simultaneously where the function f and g are two arbitrary functions, and each has a zero contour line that divides the (x , y) plane into regions where their respective function is positive or negative [Acto 90].

$$f(x, y) = 0$$

$$g(x, y) = 0$$

The solution to these two functions are points that are common to the zero contours of f and g. But we can see that these two functions have no relation to each other at all.

So in problems of more then two dimensions we need to find points that are mutually common to the N unrelated zero contour each of dimension N-1. A typical problem given N functional relations to be zeroed, involving variables

$$X_i, I = 1, 2, \ldots, N:$$

$$F_i(X_1, X_2, \ldots, X_n) = 0$$

$$i = 1, 2, 3, \ldots, N.$$

where X is the vector of value X_i and F denote the vector of functions F_i. With Newton's method, we need a good initial guess to begin to isolate the root (or roots) of functions. If the initial guess is not close to the root, then this method can not be used to find the roots of nonlinear systems of equation.

1.3.4 GRAPHICAL METHODS

Given an n equation f(x)= 0, we should obtain estimates of its roots by computing f(x) at selected points and then plotting a graph. If the function crosses the x-axis, evidently a real zero (root) exists at the corresponding points [Grav 79].

Sometimes it is more convenient to write the function in the form f (x) - g (x) = 0, rather than at the roots that is needed f = g. The estimates can be obtained by plotting both f (x) and g (x) and finding the points of their intersection.

The graphical method is primarily suitable for hand calculations or if initial crude estimates are desired for some more precise algorithm. This method is not quite practical if the interval to be tested is large or if the function is not very smooth.

1.4 THE NEED FOR A NEW METHOD

One surely can see that some pedagogical important methods, such as Bisection, secant, Newton-Raphson, Jacobi, fixed point, Baillys and many more methods for root finding (for more information on these methods or other methods that were not considered here, see the references), presume the previous availability of one or more of the following.

1- An initial estimate of a root:

2- Knowledge that a given interval contains an isolated root:

3- Details of the function, e.g. derivatives.

Even when such information is available, a chosen method may fail because of the behavior of the function near roots and in some cases the methods may rely solely upon a difference of signs of the function at the ends of an interval. Where there may be no readily apparent sign change in the function value, the notion of bracketing a root and maintaining the bracket becomes difficult. The initial estimate and derivatives are two vital parts of the above methods.

None of the methods previously discussed are actually comparable with the method that is the focus of this dissertation, in the sense that they, standing alone, do not cover as broad a range of situations. Overall, if one is given a function in a black box, there is no sure way to bracket its roots or to determine if it has roots. For the pathological examples, we will consider the problem of locating the two real roots of nonlinear system of equation which dip below zero only in very small intervals.

1.5 JONES AND IYENGAR METHOD (1994)

Recently, Jones and Iyengar have developed a complex technique for isolating the real roots of a continuous function in an arbitrary interval. In four recent publications, principles for isolating the roots of nonlinear equations have been evolved. The first three of these papers treat the case of a single nonlinear equation and introduces the use of Chebyshev's.

The research is complete for a single nonlinear equation, and a production status program for the problem has been published. The fourth paper for the root isolations extends the principles for a single

equation to straight forward manner to systems of nonlinear equations which is center of this dissertation. No algorithm which uses only function values can guarantee to isolate roots, since only a finite number of points can be employed. Moreover, since a prescribed tolerance on terminal regions is employed, more than one root may exist in a terminal region. However, we can give algorithms which isolate roots while characterizing the functions for which they might fail. By forcing the class of functions for which the algorithms might fail to have "extremes" of behavior, we gain "confidence" that we have isolated the roots.

1.6 SCOPE OF THIS DISSERTATION

The primary propose of this dissertation is to develop a computational technique by investigating a methodology of preventing the algorithm structure from converging to the same root along different paths of computation. From many possible tactics, the method chosen here complements other algorithm mechanisms.

There is an increase in computation time with an increase in the numbers of variables, an increase in computational region. For example an increase in TK (integer constant for function test), an increase in roots, or a decrease in the terminal tolerance. For this reason as well as the problem of being computationally intensive a parallel version of the program was designed and implemented on MIMD (Multi Instruction Multi Data) architecture.

Finally, the method presented here has been implemented for a real world application. The ongoing high energy physics experiment that would be use it to finding the track of neutrinos in a Super

Komiokande detector. This experiment is located in Japan and data can be processed on line or off line locally or remotely.

1.7 ORGANIZATION OF THE DISSERTATION

This dissertation is composed of three parts. The first part addresses the computational technique for isolating the root of nonlinear equations using Jones and Iyengar method. The second part proposes and implement a efficient tactic for isolating the root of nonlinear equations. The third part addresses the real world application and implementation of this metrology on MIMD architecture.

Chapter 1 introduces the problem of isolating the root of nonlinear equations and the new ideas that shows promise for isolating all the roots of relatively large nonlinear system over a given bounded region.

Chapter 2 and 3 describe and implement a efficient sequential and parallel model for the computationally intensive complex root isolating method. The effectiveness, efficiency and enhanced overall performance of the parallel processing of the program compare to sequential processing is discussed.

Chapter 4 describes the real world application in high energy physics experiment and the implementation of above program to track neutrinoes.

Chapter 5 presents a short summary of the merits and contributions of the isolating the root of nonlinear equations with parallel processing.

CHAPTER 2

SEQUENTIAL SYSTEM

2.1 INTRODUCTION

The main purpose of this chapter is to show ways to prevent the algorithm from converging to the same root along different paths. Of the many possibilities, a strategy was chosen because it could complement and integrate with other algorithm mechanisms, without complication of the problem. The algorithm then was examined using the selected strategy.

Principles for isolating the root of nonlinear equations have been evolved in four recent publications (see [Jones 77.78,84,94]). The first published paper introduces the use of Chebyshev's inequality to detect roots of a single equation. It gives an empirical demonstration of the power of the detection criterion. The paper primarily works with the anti derivatives of the function.

The second paper gives an algorithm for a single equation which employs only function values. Further insight is provided into the power of the technique, and a prototype program is demonstrated. The third publication is a production status program for a single equation; special data structures and strategy are employed to double efficiency over the earlier prototype to show the superiority of the program over the other existing approaches. The fourth paper extends the principles in a straight-forward manner to systems of equations.

This technique is used to isolate the real roots of nonlinear system of equations. The algorithm requires only function values and can be applied to complicated systems of transcendental functions.

2.2 THE ROOT ISOLATION TECHNIQUE

This procedure generalizes the methodology for a single transcendental equation. When only a single root is needed, the quickest method to provide the root is an iterative technique [Stor 80]. If an iterative technique fails, or if it is desirable to isolate more roots, the method that was proposed by Jones and Iyengar might be tried. This method has worked in all instances where it has been tried. Grid techniques often miss roots and involve an exponential increase in points with dimensionally [Rals 78]. The technique used in this dissertation provides much greater certainty, and avoids (except in worst case scenories) exponential explosions. Also, many techniques require the same number of equations as variables, a restriction which this method does not impose. However, no algorithm which uses only function values can guarantee to isolate roots of nonlinear functions, since only a finite number of points can be employed.

2.2.1 A DETAIL STRUCTURE

To facilitate understanding of this method we can look at an example of if given an set of nonlinear equations and an initial region described as

$$A_i < X < B_i$$

$$i = 1, 2, ..., n$$

then

$$f_1(x_1, x_2, \ldots, x_n) = 0$$
$$f_2(x_1, x_2, \ldots, x_n) = 0$$
$$\vdots$$
$$f_m(x_1, x_2, \ldots, x_n) = 0$$

by employing only function values this method will attempt to enclose the roots of the system in disjoint regions:

$$R_j = \left\{ (\ell_{ji}, U_{ji}) : i = 1, 2, \ldots, n \right\}, \quad j = 1, 2, \ldots, p$$

where the ith two-tuple bounds the ith coordinate. Here the intention is that each region shall be small according to a prescribed tolerance on $U_{ji} - \ell_{ji}$, and each region will contain one root. However, since the system might sometimes be zero over some type of region, we provide for a larger than prescribed final region. Further, this allows m = n, m < n or m > n, when m and n respectively are numbers of function and variables in the system.

The constituent Test (R) as shown in algorithm yields a positive indication if

$$\bar{f}_j^2 / V_j \le K^2 \qquad j = 1, 2, \ldots, m$$

where K^2 is a prescribed constant. f_j is the mean of the jth function over R, V_j is the function variance and are given respectively by

$$\bar{f} = (b - a)^{-1} \int_a^b f(x) dx$$

and the variance of $\mathbf{f(x)_j}$ from $\bar{\mathbf{f}}$ is

$$\mathbf{V = (b-a)^{-1}\int_a^b [f(x)]^2 dx - \bar{f}^2}.$$

In the next step, the algorithm will proceed by performing function evaluations until it has a sign change, or until the mean and variance estimates from function values become sufficiently stable to allow inferences to be made regarding the distribution of function values within the interval. The test (R) iteratively doubles the number of random points in R until each $\bar{f}_j^2 / V_j$ roughly converges. If the mean squared over variance test is not satisfied for every function, then a negative indication results. The basis for the mean squared over variance test is Chebyshev's inequality and empirical analysis, which shows

$$\mathbf{P\{|x-\mu| \geq k\sigma\} \leq \frac{1}{k^2}}.$$

Chebyshev's inequality and empirical analysis holds regardless of the distribution of the random variable x, where μ is the mean and σ is the standard deviation. The objectives in Partition (R) are to quickly close on roots in an uncomplex manner and to avoid a combinatorial explosion of regions when many variables are involved. The method chosen to accomplish these objectives bisected the bounds in one coordinate each time Partition (R) is executed. Each time Partition (R) is reached, the next successive coordinate is selected, cycling back to the first coordinate from the last.

The fourth [Jones 94] paper discusses a prototype program for nonlinear systems of equation. This prototype establishes the efficiency of the principles for systems; however, due to current implementation tactics, the prototype is infusible for large systems, which will bring us to the problems. The next section of this research explores and implements the tactics to remedy current shortcomings.

2.3 CORRECTNESS OF THIS PROGRAM

The essence of the root isolation algorithm can be concisely described as giving a region a root test. If no root is found, the region is deleted. If a root is indicated, the region is partitioned, and the partitions are saved. One of the partitions then becomes the region to be examined, and the process is repeated. The algorithm continues deleting or partitioning regions until a region size meets a given tolerance, in which case it is saved for output processing. Eventually, only regions for output processing remain. At this point a new technique was implemented for the correct output of the root from out stack.

The root test is central to the algorithm. Within a system, each function is tested for a root indication separately but simultaneously over a region. Sign change constitutes one root indicator; in the absence of sign change, a mean squared over variance test is employed. This test is based on Chebyshev's inequality, which can not guarantee to detect a root in a region; however, it is effective unless the behaviors of the functions are erratic. Actually, we can parameterize what this meant by erratic behavior.

By forcing the class of systems for which the algorithm might fail to have extremes of behavior, one should gain confidence that the roots are isolated.

2.3.1 COMPUTATIONAL ERROR IN ALGORITHM

As was discussed previously, Jones's fourth paper [Jone 94] extends the principles in a straightforward manner to systems of equations. A prototype program demonstrates the effectiveness of the principles in isolating roots of systems; however, the prototype is too inefficient to be practical for large systems. There is nothing inherent in the principles to generate such inefficiency, consequently the inefficiency is due to tactical considerations in overall strategy. This brings us to the focus of this research: errors in the algorithm and the resultant inefficiency they engender.

The nature of the correctness is best explicated by an example of root at the center of some region. Figure 2.1 illustrates two variables with the root in the center of the region.

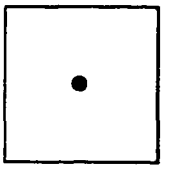


Figure 2.1: Typical region with root in the center

At the start of the search, partitioning would proceed by partitioning the region in half (r_1, r_2), repeating the partitioning until it finds the root of the function as illustrated in Figure 2.2; typically,

this procedure will take about 6 steps to find the real root of this nonlinear functions at the region's center.

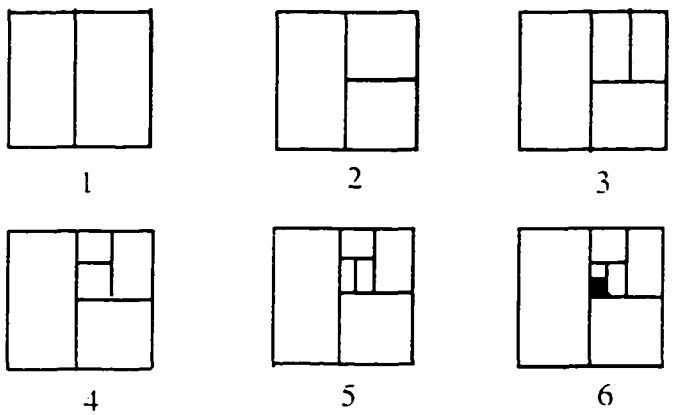


Figure 2.2: 6 step of Partitioning of the original region

Following this process, each region that was tested would be deleted from the work stack; at the conclusion we are actually left with only the regions that have the root of this function or region, as illustrated in Figure 2.3, which an out stack have only two regions.

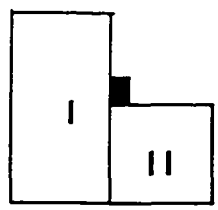


Figure 2.3: The last region

The blacked-in region represents our answer. Now regions I and II must be examined for a root. In examining region II, we obtain the same root by again partitioning each region as shown in Figure 2.4; again, this process will take about 6 steps.

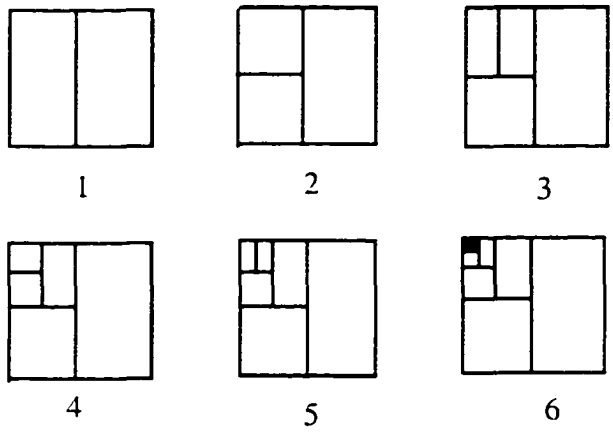


Figure 2.4: 6 step of partitioning region II

So here, we again converge on the same root blacked-in region. Similarly, when we examine region I from Figure 2.3, we will first partition it into two regions, and then we will converge to the root from each of these two regions. In general, if there are n variables, the current algorithm will converge to each root by $_2$n different paths. Here we have assumed the root to be at the center of a region; this situation is approximately replicated at some stage as we close on a root. This is especially true since Chebyshev's criterion may save a region until it is considerably reduced to the tolerance, even if a root only neighbors the region. This is one of reason that the current algorithm is infusible for large systems.

2.4 ALGORITHM

To facilitate understanding of the algorithm for nonlinear systems of equations, and the problem of correctness with the inefficiency for the prototype programs, the following simplified algorithm for finding the root of nonlinear system of equation is introduced:

1) Initial region ⟶ R

(the initial interval [a, b])

begin

1.a: for k ← 1 to n do

if the function suffers a sign-change then

go to 3 else

compute running sums of the function value and its square

compute v and f

endif

if v and f have stabilized then

go to 2.c else

set n ⟶ 2*n and go to 1.a

endif

enddo

2) Test region(R)

2.b: If Negative, then

Delete R and Proceed to 3.

2.c: If Positive, then

Partition (R) into r_1 ,r_2.

Put r1 and r2 on to stack

endif

endif

3) Pop work stack ⟶ R (if empty go to 5.)

4] Check Size of (R) < tolerance (e of program)

4.a: if no, go to 2

4.b: if yes, R ——→ outstack, go to 3.

endif

endif

endfor

5) Return region in out stack.

Consolidate adjoining intervals

6) Check the outstak for number of roots

if number roots < or = 1 then go to 7

else scan through output stack and consolidate any adjoining intervals.

if they contain the same root, and

if the roots are equal

and<tolerance, then

set the flag

endif

endif

endif

7) Delete the same root from out stack

end

2.4.1 ANALYSIS FOR ERROR IN PROGRAM

In analyzing the original algorithm and program for possible improvement for converging to the same root by 2^n, several errors

were found. The first error was that kk=0 in the main program was always set to 0 and never were changed. Also the xmean(I), X(I), initials were noted by very small letters; additionally, double precision was not used in the module root. All these errors caused the program to fail to capture roots of nonlinear systems of equation. The errors were detected while using the test functions. The errors were removed, and the corrected program is free of those errors.

The next error was convergence to the same root by 2^n. Keeping in mind that the solution should converge to the same root by different paths and should complement and integrate with other algorithm mechanisms, the steps (6 , 7) in algorithm above were added to the original algorithm. Where this solved the problem of a duplicated root in the out stack or output file. This tactic did not complicate the problem of the root-finding algorithm and program. The abstract FORTRAN code for this part of algorithm, developed and implemented into the root isolation program, is shown below. The solution for the implemented technique discussed above works as follow.

1-The outstak is scanned and checked for the number of region or roots that were saved.

2-Next if the number of roots are more then one, the program was checked for coordinates.

3-A flag is set for all roots with the same coordinate.

4-One of the two roots with same flag is deleted and output is generated.

The updated FORTRAN program used to implement the above technique is shown below. For a more complete list of arguments and the code, see appendix A.

```
c**** check for numbers of roots
  200 if (nroot.eq.0)        go to 750
      if (nroot.eq.1)        go to 290
      if (nroot.gt.1)        go to 210
c**** check for root in the same region.
  210 kk=0
      do  i1=1,nroot-1
            do  i2=(i1+1),nroot
                  do  j=1,n
                        kk=kk+1
                        diff(kk,1)=abs((outstk(j+n*(i1-1),1))
     $                  -(outstk(j+n*(i2-1),1)))
                        diff(kk,2)=abs((outstk(j+n*(i1-1),2))
     $                  -(outstk(j+n*(i2-1),2)))
                  enddo
            enddo
      enddo
c**** set flag for same roots
      do  kk=1,n
            do  i=1,2
                  if (diff(kk,i).gt.(1.e-2)) go to 290
                  if (diff(kk,i).le.(1.e-2)) jjj(kk)=1
                  if (jjj(kk).eq.1)  nroot=nroot-jjj(kk)
```

```
            enddo
        enddo
        if (jjj(kk).eq.1) then
        nroot=nroot-jjj(kk)
        endif
```

2.4.2 IMPLEMENTATION

The idea presented in the previous section was implemented on a VAX 3100 workstation running VMS 5.1 operating system. With this very simple, but effective, technique the updated program detailed above was tested with two sets of nonlinear equation systems. In the first case, we examined two functions of the system, and in second case three function in the system equation were examined.

The result from these tests are shown in Table 2.1 and Table 2.2. As can be seen from Table 2.1, two roots had the same region information, (X(1), X(2) and X(3) are the three variables) in the original functions. The algorithm converges to this region (root) from two different paths. With the new technique implemented, the new program was able to correctly pick the real root of the nonlinear system of equations, and also it was able to select and delete the root that was reached by an error.

The result in Table 2.2 also confirms the fact that the above new algorithm can find the extra root that was reached by an error, and eliminates the problem of two roots reached from different paths. Table 2.2 shows that there were three outputs for X(1), X(2) and X(3) with the old program, but the new program was able to pick up the true real roots of nonlinear functions and delete the one root reached by error.

The results of this new program can be seen by the numbers of roots that were isolated by the program in Table 2.2.

Table 2.1 : Root isolation output for 2 functions

TOTAL ROOT FOUND =1					
X(1)		X(2)		X(3)	
FROM	TO	FROM	TO	FROM	TO
0.914063	0.945313	1.000000	1.007813	0.984375	1.015625
0.914063	0.945313	1.000000	1.007813	0.984375	1.015625

Table 2.2 : Root isolation output for 3 functions

TOTAL ROOT FOUND =2					
X(1)		X(2)		X(3)	
FROM	TO	FROM	TO	FROM	TO
1.531830	1.532440	1.8788967	1.879578	0.4672224	O.468750
-2.000732	-1.999512	-0.000916	0.000916	3.999023	4.001465
-2.000732	-1.999512	-0.000916	0.000916	3.999023	4.001465

2.5 PROOF OF TERMINATION

The law of large numbers and the fact that a region is deleted or reduced to smaller regions (each smaller than the previous region by at least a fixed region size) can be invoked to show the algorithm terminates. If we let R(1), R(2), ..., R(m) be the interval in the stack at an arbitrary time, where I(k) was pushed before I(1) if k<1.

This set is disjoint and R(1) < R(2), ..., < R(m). This is easily shown by induction on the number of times the successful part of step test (R) is executed. If this step is executed only one time, then the subdivision procedure in next two steps will guarantee statement truth.

This will prove the above guarantees that if an interval is discarded or put on the output stack the new test (R) will not be performed on it . This meant we can be sure that the algorithm will never go into an infinite loop. This program has been successfully employed on numerous nonlinear systems which are not shown in this result, in all test cases the program correctly isolated all the roots. However, in some cases it was necessary to change the constant parameter c (the value of K^2) until all the roots were found.

2.6 POTENTIAL OF DERIVING A PARALLEL SYSTEM

The performance of sequential program included metrology of preventing the algorithm structure from converging to the same root along different paths of computation, proves the fact that this algorithm is unfeasible for large system of equations. The algorithm is so computationally intensive that the system is limited to very small set of variables.

A more realistic solution to this problem is implementation of the sequential program on parallel system. This will give good potential for real time application and it should reduces the cost of execution.

CHAPTER 3

PARALLEL SYSTEM

3.1 PARALLEL MODEL

Before parallel version of the root isolation program was designed and implemented the parallel model of computations architecture and method of designing parallel algorithm were defined and are presented briefly here in this section.

Parallel processing is information processing that emphasizes the concurrent manipulation of data element belonging to one or more processes solving a single problem and parallel computer is a multiple-processor computer capable of parallel processing.

Since the design and performance of a parallel program depends on the architecture of parallel system, it is important to keep the architecture in mind. Consider the number of different architecture in market today, one selected here was best for the need of computationally intensive parallel version of the program.

With MIMD (Multi Instruction Multi Data) architecture each processing elements (PE_S) are individually indexed and have some local memory [Iyeng 86]. The PE_S operate synchronously under the control of individual instruction streams where each processes can execute it's own instructions at any time.

Message passing is the means of communication among processes and in many MIMD models the communication time often

dominates the overall complexity and execution time of program. An example of this architecture is IPSC/860 where it was used here for implementing parallel version of the root isolation program

Many problems can be solved by massive parallelism, however many algorithm suitable for conventional, single-processor computer are not appropriate for parallel architecture. There is no universal method for designing parallel algorithm but one approach is to recognize parallelism in the existing sequential program.

The sequential program for root isolation it have inherent parallelism in form of repeated iteration and use of stack which each segment can work at full speed without data dependency where each PE_S can individually with minimum communication complete it's own task.

3.2 INTRODUCTION

In analyzing the original program for possible improvement, the main concern was to reduce the run-time of this program. Generally, there was an increase in CPU time as variables, regions TK, or roots increased; CPU time also increased as the terminal tolerance decreased.

As shown in Figure 3.1, when the program was tested with 2, 4, and 6 functions and the same numbers of variables, the run-time changed about e^2 . Figure 3.1 illustrates the extent to which the CPU time depends on number of functions and variables in the system.

In the Jones paper [Jones 94] a test was conducted to show the effectiveness of the algorithm. In one of the tests there were six functions with six variable nonlinear systems of equations, which required 6 minutes to isolate two roots. Now, if one were to divide the 6

minutes by 2 (2^6 - 1), the result would be approximately 3 seconds. Of course, we would probably not obtain this speedup with a normal sequential algorithm and system, but the 3 second gain in sequential algorithm should be roughly indicative of the possible improvement that can be applied to this algorithm.

The speedup suggests that if the shortcomings of the algorithm could be remedied, much larger systems can be solved in reasonable time with today's technology in computer hardware and software.

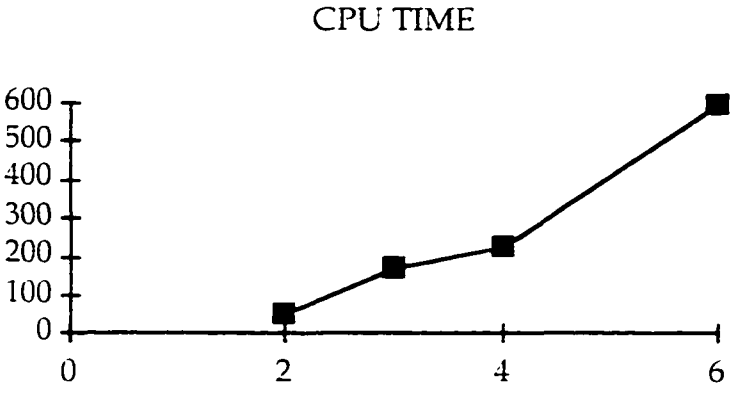


Figure 3.1 Number of equation and variable

As the systems become more complex or larger, program run time is dominated by system evaluations. Referring back to the algorithm of root isolation for systems of nonlinear equations from the previous chapter, one can see that Test (R) is executed, and the region is subsequently divided into two regions upon a positive indication. These two regions are later subjected to examination by Test (R). Test (R) involves the computation of the system values, and an examination of the sub regions which requires that values in the

region be computed again. As regions are repeatedly tested in this manner, regional recompilations become numerous. For this reason, a parallel version of the algorithm is very much needed, and consequently it was designed.

3.3 SEQUENTIAL PERFORMANCE

A parallel program of the algorithm is very much needed in order to be able to isolate the roots of large systems of equations. In order to learn more about the problem, the program and it's modules will be presented in this section. Also we will discus the performance of sequential program which was analyzed.

As described in Appendix A, the root program consists of four modules: the main, root, point and system modules. The main module is mostly used for input, output management and to facilitate user interface of the program. Also as previously discussed in the last chapter the main module is responsible for multiple roots in the same region.

The root subroutine is used to test (R) and cut the interval; this subroutine is also used to work the two stacks (work stack and out stack) in which all intervals are saved for further analysis. The point subroutine is used to do all the book keeping and transfers of function value. The last subroutine system is used as a Monte-Carlo random number generator. The function of this subroutine is to generated the value for each variable and passed on to the point subroutine. At the module point, function values are determined, and some book keeping is performed before passing these values (functions, variables) back to the root module. When the values reach the root modules these

values are used to test each region. The performance and run-time of the sequential root isolation program was analyzed with the set of test functions that were used in the pervious chapter, and the following results shown in Table 3.1 were obtained.

Table 3.1 Performance Analysis

NAME	TIME USED
Main	1.6%
Root	30.4%
Point	44.9%
System	21.5%

As shown in the above table, most of the program's run time is expended in the point subroutine. Since the system subroutine also is related to this same point, then we can say that over 65% of program run-time is spent in the last two subroutines. The bottleneck which occurred at the root subroutine used about 30.4% of run time, mostly to test the functions for roots and to put unexamined partitions in the work stack or move them to the out stack for output.

Understanding the condition under which they occurs makes bottlenecks easy to predict. Typically each work stack (wrkstk) contains unexamined partitions to be extracted and tested. Each of these regions are extracted and tested one by one; therefore, multiprocessor configuration implementation in the work stack could reduce the run-time of program by using the regions that are saved in the work stack. The effectiveness, efficiency, and speedup of the overall performance

of each subroutine can be increased if the main loop of algorithm uses the 65% of run-time saved through parallel processing.

3.4 IMPLEMENTATION

The parallel computing facilities at Louisiana State University (LSU) were first installed in the Fall 1991, and located in Room 122 of Nicholson Hall. There are two parallel computers in this facilities: (1) Mas Par MP-1, which is a Single Instruction Multiple Data (SIMD) machine; and (2) the Intel IPSC/860 which is a Multiple Instruction Multiple Data (MIMD) machine. Originally the IPSC had 8 nodes, but due to hardware problems, only four of these nodes are usable at this time. The processors are cube connected (hypercube). The peak floating point performance rating of this machine is 640 Mflops at 32 bit, and 480 Mflops at 64 bit. Each node has a local memory of 16 Mbytes. The IPSC/860 (MIMD) computer was used for the implementation of the parallel program. Each processor works with different sets of data independently of others.

In order to implement the ideas presented in the previous section, the following hardware and software were needed:

1. IPSC/860 (MIMD Machine)
2. Message passing model.
3. FORTRAN 77/90 compiler

The changes/additions that have been made to the original program will be discuss in detail in the next section.

3.4.1 IPSC/860 (MIMD MACHINE)

In this section, we introduce the following IPSC system components used to implement the program. System hardware (host,

nodes, and peripheral units); host and node system software; Development environment (languages); Runtime environment (system commands and system calls) were necessary to implement to program. For more detail information on IPSC/860 and it's components see Appendix B.

3.4.1.1 THE HOST

The host provides access to all the nodes. One communicates with the nodes through a front-end processor called the host; there are two kinds of hosts. The first kind occurs when an IPSC system comes with a local host SRM (the system resource manager) connected directly to the IPSC nodes. The system can be configured by workstation (by installing remote host software) to serve as a remote host. A remote host can then be connected to the SRM over an Ethernet link. This was the option that we selected in order to facilitate access to the IPSC.

The host (local or remote) runs the UNIX operating system and provides the interface between user and the nodes. In the typical IPSC model, the program can be compiled and link the host and node programs on the host; using this model, we then loaded the executable to run the host programs on the host and the node programs on the nodes.

3.4.1.2 THE NODES

A node is a processor/memory pair. Each node's memory is distinct from the host and from other nodes. Each node runs the NX/2 operating system and communicates with the other nodes by passing messages. The collection of nodes belonging to an IPSC system is called a cube. The number of nodes in a cube is defined by its dimension

(expressed as dn). For example, a d7 cube has 2^7 (128) nodes. Each node is fully connected to every other node through its Direct Connected Module (DCM) which has 8 channels. Channels 0 through 6 are connected to the node's nearest neighbors. Because of the DCM, there is no "store and forward" option for messages. Nodes that are close neighbors have nearly the same message latency as those that are not.

The system has four kinds of nodes: Compute nodes, I/O nodes, Integrated nodes, and Service nodes. Compute nodes are designed to perform computational tasks. Although there are five kinds of compute nodes, this dissertation is limited to the one that was used in this study. The CX nodes are based on the Intel 386 processor and have an 80387 numeric coprocessor. The SX nodes are CX nodes in which the 80387 numeric coprocessor is replaced with an SX processor (for doing high-speed, floating-point, scalar arithmetic). The RX nodes are based on the Intel 860 microprocessor. CX, SX compute nodes are often collectively referred to as CX nodes, because they are all based on the Intel 386 microprocessor.

3.4.1.3 IPSC SYSTEM SOFTWARE

The IPSC system software consists of the Host operating system software; the Node operating system software; the Software development environment; and the Runtime environment. The Host operating system software (the host local or remote) runs the UNIX operating system with IPSC extensions and TCP/IP networking software. The IPSC extensions include commands, libraries, and background processes that support communication with the nodes. The TCP/IP networking software links the SRM with remote

workstations. The IPSC extensions consist of IPSC system commands, system calls, application libraries, and background system processes. Together, these extensions provide the interface to the cube. To provide access to the cube and support communication between the host and the cube, the host runs the IPSC system processes (in the background). The SRM and remote workstations also run the TCP/IP networking software, which allows the user to log into other systems on the network remotely and to transfer files between systems.

3.4.2 MESSAGE PASSING

Message passing is the means of communication among processes in an IPSC system. The message passing model was used here to implement the new algorithm. As independent processor/memory pairs, the nodes do not share physical memory. In addition, even node processes running on the same node maintain distinct memory spaces. If the node processes need to communicate, they do so via message passing. Message passing calls can be synchronous or asynchronous.

3.4.2.1 MESSAGE CHARACTERISTICS

All the messages were characterized by a length, a type, and an identification. Then each message length is measured in bytes. The message-sending routines were used to send exactly the specified message length, and the message receiving routines were used to place no more than the specified length into the receive buffer.

If the receive buffer was not large enough to hold the message, an error resulted.

As shown in Table 3.2, the following calls show the Synchronous and Asynchronous Message Passing Calls for both the

nodes and host. For example, csend can send a message with sending node identification and receiving node identification which it has exact length of the message.

Table 3.2 Massage passing calls

Call	Environm.	Description
crecv	host/node	Synchronous receive
csend	host/node	Asynchronous send
irecv	host/node	Asynchronous send
isend	host/node	Asynchronous receive
isendrec	host/node	Asynchronous rec/send

3.5 A CASE STUDY

Concurrent applications have varying degrees of parallelism. A perfectly-parallel application is one that requires no node [Hell 78] communication. In a perfectly-parallel application, if one were to double the number of nodes, this would cut the computation time by approximately 50%. But in real life most applications involve a mix of computation and inter node communication.

Our goal for this problem was to develop a communication strategy that maximizes the time a node spends computing and minimizes the time it spends communicating. The concurrent programming model that we designed needed to have the following characteristics.

1. An ensemble of processor/memory pairs called nodes that make up a cube; They do not share memory. They can access the same file system, but they operate independently of each other.
2. All the nodes are fully connected. They communicate with each other and the host by passing messages.
3. Each node executes its own program on a different set of input data. There may be some conditional code that identifies one or more nodes that perform special actions.
4. The goal was to keep all the nodes busy and have them finish at the same time.

Based on the parameters mentioned above a parallel program was developed from exciting sequential program that had all the characteristics of the previous discussion. The first part of the original program that was modified it was the work stack (wrkstk), where we were able to implement the new parallel algorithm and look at the problem of efficiency time used by the stack.

3.5.1 TEMPORALLY BUFFER (WORK STACK)

As was mentioned previously the region is subsequently divided into two regions upon a positive indication, when the Test (R) is executed. These two regions are later subjected to examination by Test (R). One of the setbacks in the old program was that each partition had to be kept in the work stack in order to be examined sequentially as shown below. The work stack was opened and a new region was partitioned in sequence. Here the INDEX is an integer, which shows the number of regions in the stack and ID identifies the dimension that

is partitioning currently. The n is number of dimensions (variable).

```
Begin
      do  i=1,n
            ie=(index-1)*n+i
            a(i)=wkstk(ie,1)
            b(i)=wkstk(ie,2)
            wk=wkstk(ie,3)
            id=ifix(wk)
      enddo
end
```

One can see clearly that if the index is more then one, then there is more then one region on the work stack. So each partition can be assigned to different nodes in a parallel system. This was possible with IPSC which had more then one node in each region and, there for, could be selected by different nodes simultaneously. After selecting, the IPSC, part of original code was modified to new parallel code below. Where "iam" is a node id and "gcol, gcolx" are global call.

At the if statement, each node collected one partition which corresponds to the node's id, and at the end of the run the result from each node was collected in subroutine (gcol). The total results were put together by (gcolx) which is the contribution of each node.

```
begin
      do  i=1,n
            if(iam.ne.mod(index,4) )  then
                  iam=iam+1
```

```
                ie=(index-1)*n+i
                a(i)=wkstk(ie,1)
                b(i)=wkstk(ie,2)
                wk=wkstk(ie,3)
                id=ifix(wk)
                nbytes=4*nroot
                nodes=numnodes()
                call gcol(nbytes,4,lens,4*nodes,ncnt)
                call gcolx(outstk(1,1),lens,glo_outstk(1,1))
                call gcolx(outstk(1,2),lens,glo_outstk(1,2))
        endif
    enddo
end
```

3.5.2 PARALLEL RANDOM NUMBER GENERATOR

The next part of program to be modified was the system subroutine in which the Monte-Carlo was used to generate random numbers; later, these numbers were used to determine the function values. The following is the original section of the code.

```
170 do 175 j=1,n1
        new=nold*65539
        if(new)125,135,135
125             new=new+2147483647+1
135             s=float(new)
                s=s*(0.4656613e-9)
                nold=new
175 enddo
```

The Monte-Carlo used a startup seed to generate new numbers. As one can see with the above code we can not generate more then one set of numbers (numbers for each variable) each time this subroutine is called. We needed to get each processor to work independently of all other processors proctors for two reasons: to reduce the number of messages passing among the nodes; and to make this part of the program produce individualized random numbers each node. In order to do this we had to work with the same seed and generate numbers that corresponded to the number of nodes or processors and then to make each processor collect its own generated number correctly. Keeping these restrictions in mind, the following changes were made to the sequential program.

```
Begin
iam=mynode( )
nodes=numnodes( )
if( ifb.eq.1 ) go to 170 else
   k=0
      do  i=1,nodes
            do  j=1,n
                  new=nold*65539
                  if(new)120,130,130
                  120 new=new+2147483647+1
                  130 r(k+1)=float(new)
                  r(k+1)=r(k+1) *(0.4656613e-9)
                  k=k+1
                  nold=new
```

```
            enddo
        enddo
c**** obtain random points in region
c
do j=1,n1
        x(j)=r(iam*n1+j)
        x(j)=(b(j)-a(j))*x(j)+a(j)
enddo
x ( j ) = s
x ( j ) = (b ( j ) - a ( j )) *x ( j ) + a ( j )
end
```

When the above changes were made, we were able to get each node to work independently in different sets of the region. When the first processor got to this part of the code it continued by generateing numbers for itself with the old seed; it also generated numbers for all of the other processors.

Then these numbers were saved in a global area to which all other the processors had access. When the lagging processors arrived at this point, the lagging processors will recognize the flag set for them. Each processor will look in the specific area in global area, and will collect the numbers flagged to their id number. At this point each processor has distinct regions to work with and have no need of other processor output or input; each node can execute its own data.

Table 3.3 shows the result of this operation for sets of numbers that were generated with this method for parallel number generating routing. The results for four nodes are compared to the results derived

from the sequential output of the same routing with only one node (sequential).

Table 3.3 Monte-Carlo result

Sequential		Parallel	
node #	Randm point	Node#	Randm point
1	676165863	1	676165863
1	1909418677	2	1909418677
1	1076051999	3	1076051999
1	89621141	4	89621141
# of irritation in loop = 4		# of irritation in loop = 1	

As we can see, using the sequential system we needed four loops to generate the random point in the region, but in the parallel system all we needed was one iteration for all random points.

3.5.3 ROOT MODULE

The next changes were in the root subroutine, which is the heart of the root isolation method. Since many changes were made to this part of the program, only the most important and effective changes will be shown here. The rest of program can be seen in Appendix A.

One of the most important parts of the root isolation method is the test (R) by which the functions are tested for sign changes by calculating the means and variances for each equation. In order to do this the function value must be generated.

This was repeated approximately 10 times for each function evaluation. With the power of parallel programming, this is possible in 10/n where n is the number of nodes allocated. With this in mind

the flow diagram of root isolation programs was changed and the resulting flow diagram is shown in Figure 3.2.

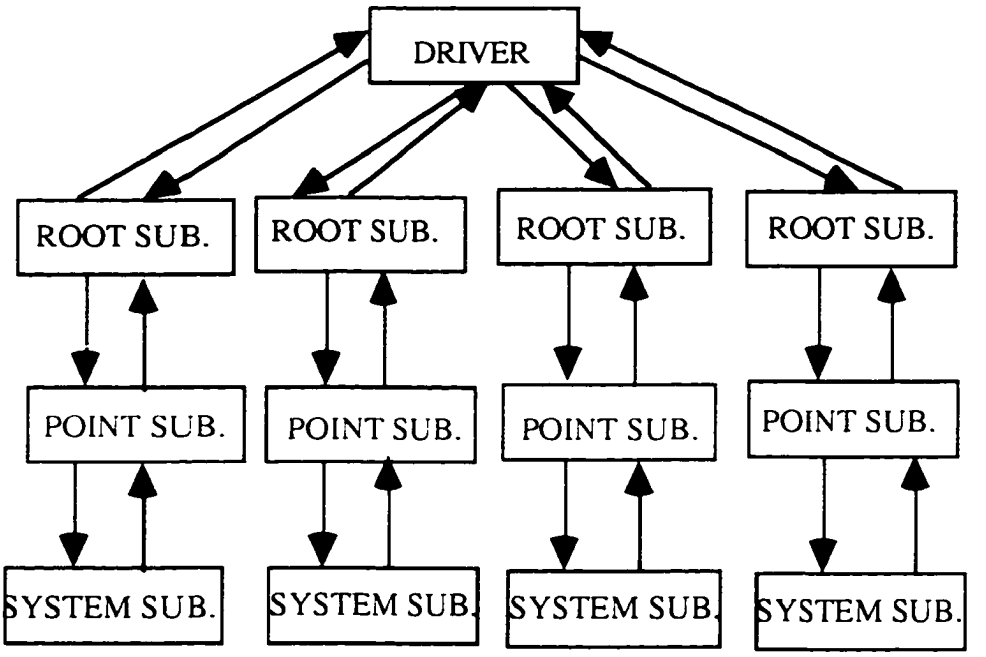


Figure 3.2 The structure of the parallel root isolation package

The original code was changed so each node can take part in the iteration and return part of the result. At the end of iteration, the results from each node were added together; then the overall result of iteration is put into global area where all nodes can have an individual copy. The section of code that implement the above metrology and divided the program for each iteration from among all the allocated nodes is as follows:

```
begin
npn=k/nodes
imin=(npn*iam)+1
imax=imin+(npn-1)
```

```
if( iam.eq.nodes-1) imax=k
do 90 i=imin,imax
        call points(b,a,ff,nx,nz,n,m,ifb)
        do 80 l=1,m
                sum(l)=sum(l)+ff(l)
                sum2(l)=sum2(l)+ff(l)*ff(l)
        80 continue
90 continue
call gdsum(sum,m,temsum)
call gdsum(sum2,m,temsum2)
end
```

3.6 SPEEDUP

There are three speedup models which are used in parallel computing today [Quin 94]. Each one of these speedups is used under different computing objectives and resource constraints. These include Amdahl's law, Gustafson's scaled speedup, and the memory-bounded speedup. The fixed load model (Amdahl's law) corresponds to the type of constant workload which eventually communication bound limits the use of this model.

The fixed memory model is limited by memory bound which maximize the use of both Amdahl's law, and Gustafson's. The idea is to solve the largest possible problem, limited by memory space. This also demands a scaled workload, providing higher speedup, higher accuracy, and better resource utilization. The fixed time model (Amdahl's law) demands a constant program execution time, regardless of how the workload scales up with machine size.

This model may also be refereed to as linear workload growth because work load growth is linear to machine size in order to keep the time constant. One of the major shortcoming in applying Amdahl's law is that the problem cannot scale to match the available computing power as the machine size increases. In other words, the fixed load prevents scalability in performance as shown here.

S_n= Generalized Speedup = Parallel Speed / Sequential Speed

Amdahl's law $\sum_{i=1}^{m'} W_1 / \sum_{i=1}^{m'} Wi = T'(1) / T(n) = S' n$

3.6.1 GUSTAFSON'S LAW

For the reasons stated above, the fixed time model (Gustafson's law) was selected to analyze the speedup performance of this parallel application in which the sequential application of the run-time and problem size was problematic. As the number of functions grew in our sequential applications so, did the run time. With this parallel application the goal was to solve the largest size problem possible on a larger machine with about the same execution time required to solve a smaller problem on a smaller machine. With increases in the machine size, an increased workload resulted which necessitated a new parallelism profile. Figure 3.3a demonstrates the workload scaling situation, and Figure 3.3b shows the fixed time execution style.

To sum up all three speedups and look at our computing objective, it is easy to see that fixed time speedup is best suited for this problem. Fixed load and memory bounded speedup were not related to our problem, since the need for scaled system size was the goal.

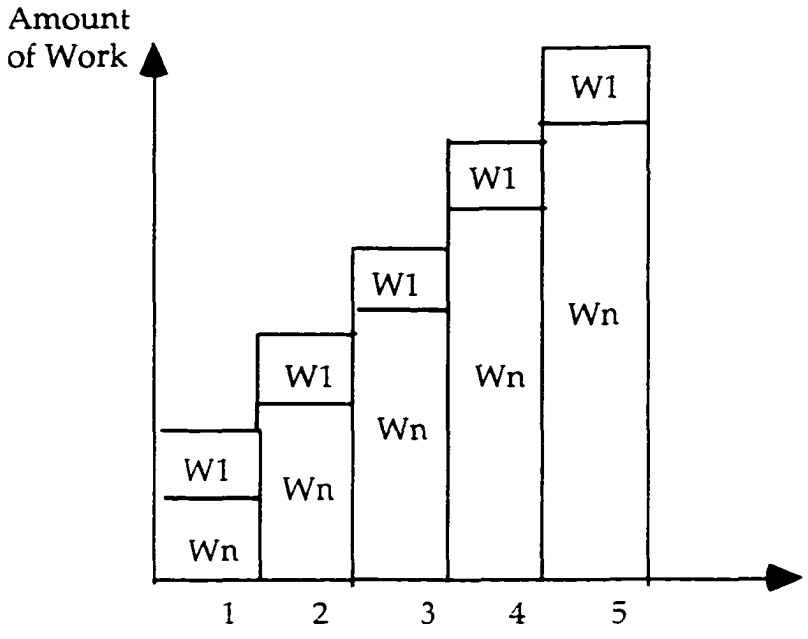


Figure 3.3a Scaled workload

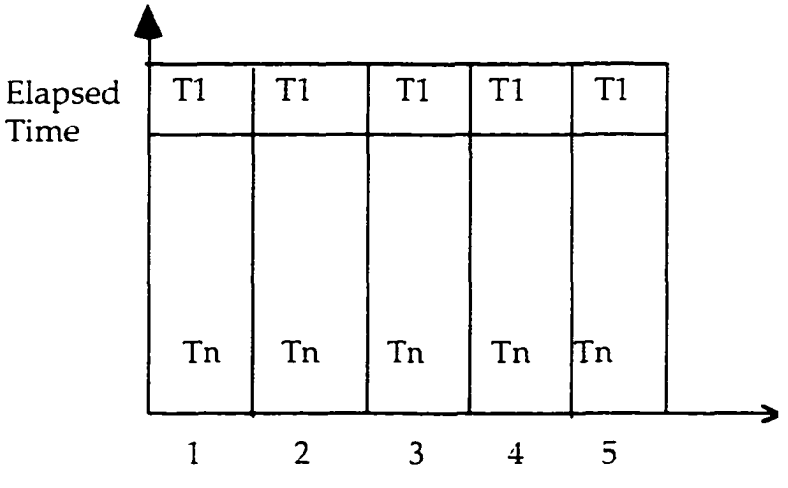


Figure 3.3b Fixed execution time

A general formula for fixed time speedup is defined by [Hatc 91]

$$\sum_{i=1}^{m'} w'_i / \sum_{i=1}^{m'} w_i = T'(1) / T(n) = S'.$$

where m' is the maximum degree of parallelism (DOP) with respect to the scaled problem, and w' is the scaled workload with DOP = i. In general, w' > w for $2 < i < m'$ and w' = w. Also T'(n) is the execution time of the scaled problem, and T(1) corresponds to the original problem without scaling.

3.7 RESULT AND ANALYSIS

After all the parallel modules were developed and put together, the program was tested for correctness and effectiveness. The results can be seen in Tables 3.4 and 3.5 which compare the run-time. The success is due to modification of the sequential program to a parallel program and being able to reduce the over head of message passing. The new parallel program was tested and run under the following conditions:

1. Test functions were chosen which cover a reasonably wide range of situations.
2. Parallel run times and sequential run times were measured by the same system(IPC860).
3. The same set of functions and parameters were used in all tests for different processor runs.
4. The same compiler and resources were used for all tests.

A sample output that was generated is shown below, and a complete set of all the outputs is shown in Appendix B. The abstract results of all the runs are shown in Tables 3.4 and 3.5 which compare the run-time for condition using one, two, and four processors to those where only four processor were available with (IPC860).

```
tk=.6, A=-10, B=10, nodes=1
F(1)=X1-X2**2+2    F(2)=-X1**2+X2+X3        F(3)=2*X1-X2**2+X3
      **************************************************
    * CPU TIME FOR THIS SET OF FUNCTIONS IS   8.072931 *
      **************************************************
                NUMBER OF ROOTS ARE        3
                NUMBER OF NODES ARE        1
      **************************************************
                ROOT BOUNDARIES FOUND
                X( 1) =   1.530762  TO   1.538086
                X( 2) =   1.877441  TO   1.884766
                X( 3) =   0.458984  TO   0.488281
                ROOT BOUNDARIES FOUND
                X( 1) =   0.341797  TO   0.356445
                X( 2) =  -1.538086  TO  -1.528320
                X( 3) =   1.645508  TO   1.660156
                ROOT BOUNDARIES FOUND
                X( 1) =  -2.011719  TO  -1.992188
                X( 2) =  -0.019531  TO   0.009766
                X( 3) =   3.984375  TO   4.042969
```

Table 3.4 parallel time for tk = 6

NODES	NUMBERS OF FUNCTIONS			
#	2	3	4	6
1	2.325094	8.072931	34.85888	275.2281
2	0.964323	5.399231	11.34062	100.1219
4	0.726655	3.978216	10.28291	82.53902

Table 3.5 parallel time for tk = 7

NODES	NUMBERS OF FUNCTIONS			
#	2	3	4	6
1	3.579084	-	-	-
2	1.008329	-	-	-
4	1.078191	-	-	-

After the results were obtained for both cases of constant numbers, the tk=6 and tk=7 (tk=7 only was run with first set of function in the system) as shown in Tables 3.4 and 3.5, then the speedup for each of the above test runs was calculated using Gustafson's formula. These results are shown in Table 3.6. As shown in Table 3.6, we were able to achieve a significant speedup in all the test runs which proves the point that parallelization of the original program reduces the run-time by 1/3. Where this was our goal.

Table 3.6 Parallel Speedup for tk = 6

NODES	NUMBER OF FUNCTIONS			
#	2	3	4	6
1	1	1	1	1
2	2.4111	1.495200	3.07380	2.74873
4	3.119972	2.02928	3.38998	3.33452

Table 3.7 Parallel Efficiency (The generalized efficiency for parallel system is defined by GE= speedup / number of nodes.)

NODES	NUMBER OF FUNCTIONS			
#	2	3	4	6
1	-	-	-	-
2	1.2055	0.7475	1.535	1.37443
4	0.7775	0.50732	0.84749	0.83363

CHAPTER 4

A PHYSICS APPLICATION

4.1 INTRODUCTION

In the first section of this chapter we will discuss the neutrino and various types of devices to detect this particle, including the data acquisition and transport of data. The remaining sections of this chapter are devoted to discussion of the various types of apparatus used to detect the presence of charged particles. The use of root isolation technique to track this particles will be discussed also.

For various reasons (including safety) experiments are usually [Perk 89] performed at some distance from the accelerator, which makes it necessary to convey the beam of particles across the required distance without undue loss of either energy or intensity. The equipment used for this purpose of beam transport often must fulfill other functions at the same time; for instance, an experiment at a proton synchrotron may require a beam consisting of particles that are unstable (such as neutrinos) or that do not lend themselves to easy production in the ion source of the accelerator (e.g., positrons). Once the particles have left the general region of the detector, experimental equipment is needed to detect the presence of particles and to count how many of a given type are involved. Equipment is also needed to measure the momentum, velocity, and energy of the particles.

A wide variety of equipment is available for these functions and will be described in subsequent sections of this chapter. It will be noticed that the various particles need various methods like software, hardware, and tools to be detected.

Detectors are normally only sensitive to electrically charged particles; for example, although the neutral particles can be detected, they can only be indirectly detected rather one should observe the charged particles that result from the decay of the neutral or from its interaction with matter.

In any nuclear experiment it is important not only to count the number of particles, but also to be able to identify the particles that were counted. Many schemes have been used successfully, and I shall not attempt to list them all. Rather, I will mention a few principles on which some of the methods depend. So here we will concentrate on identification of charged particles.

In order to identify a particle, it is often adequate to measure its mass. In macroscopic physics, mass measurements usually depend on gravity in some way, but for nuclear particles it is necessary to use other methods because electromagnetic effects are so much stronger than gravitation. It is therefore necessary to use less direct methods. If the momentum (p) and the total energy (E) of a particle can be measured independently, the mass can be obtained from the equation

$$m^2c^4 = E^2 - p^2c^2$$

Alternatively, it is sufficient to measure either the momentum or the energy and combine this information with a measure of the velocity.

In order to understand some of the methods used to measure energy or velocity, we need to consider the mechanism by which energy is lost as a charged particle travels through matter. As we already know, charged particles collide with atomic electrons and remove them from atoms. If we consider a particle traveling in the x direction with a kinetic energy, then the energy loss is usually written as

$$-dE_k / dx \approx 1 / \beta^2$$

where ß=v/c. Figure 4.1 shows a graph of energy loss as a function of momentum for several kinds of particle in liquid hydrogen. This figure demonstrates that the energy loss decreases with increasing momentum until it attains a minimum value.

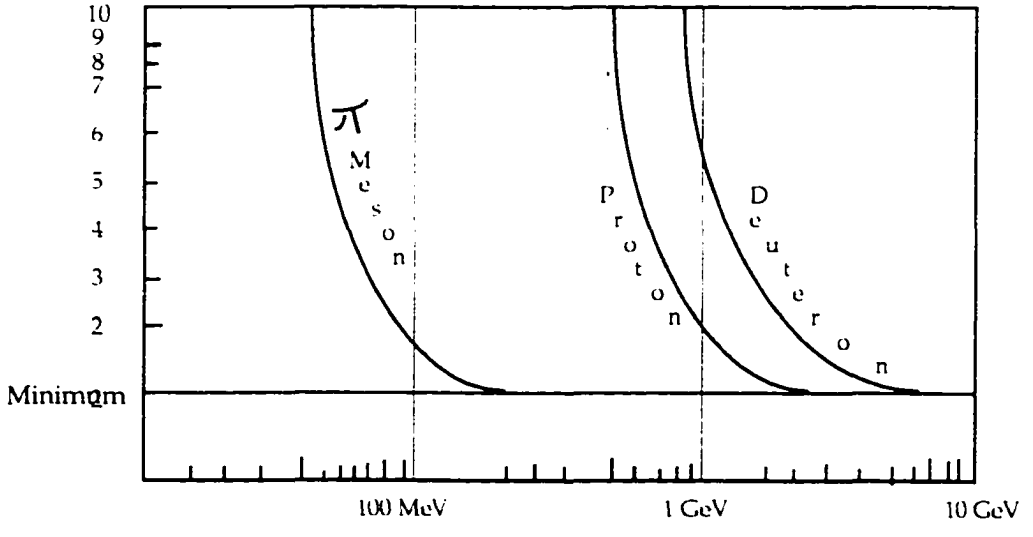


Figure 4.1 Energy loss in liquid hydrogen

4.2 CERENKOV LIGHT

In general, any electrically charged particle (with sufficiently large energy) traveling in water produces Cerenkov light (see Appendix C). More precisely, the light is generated when a particle moves with a speed greater than the speed of light in water. But this speed, of course, is always less than the speed of light in a vacuum. In a vacuum the particle generates something analogous to the shock wave generated by supersonic aircraft in the air. The 'optical shock wave' light is emitted in a cone [Bark 83].

When the cone of the light reaches the wall of the detector, it forms a ring-shaped pattern on the wall. The photomultipliers (see Appendix C) can detect the light when it arrives at the wall. Then the signal from photomultipliers are measured by other devices like the Analog Digital Converter (ADC), which indicates how much light arrived and the time of arrival (see Appendix C).

4.3 NEUTRINO

The neutrino is a light which it may be massless, and it is neutral. This means that it has no electrical charge, and neutral particles are virtually non-interacting with matter. Millions of millions of neutrinos cross Earth each second, but only very few of them interact with Earth. In practice one may say that they are invisible.

There are three types of neutrinos in nature: electron, muon and anti-neutrinos. There are many sources of these particles; each time there is a transformation between protons and neutrons, neutrinos are released. At that transformative moment, a huge number of neutrinos

are produced which are detectable on Earth just as nuclear fusion (see Appendix C), super-novas, (see Appendix C), the sun, and the earth atmosphere by which they are produced can be detected by energetic cosmic ray particles. After entering the atmosphere, the particle interacts with air atoms and produces several other particles, which in subsequent interactions with air produce even more particles, and so on. This is called a cosmic ray shower. Some of the produced particles are unstable and most of the unstable particles are pions. These decay into muons, which then decay to electrons. Since electron are stable, they don't decay any further. At each decay neutrinos are produced: electrons, muon-neutrinos, and anti-neutrinos.

Another source of neutrinos is the nuclear reactions which occur inside of our Sun. These nuclear reactions occur on a gigantic scale. So enough neutrinos are produced that they can reach the Earth and be detected. But we see too few solar neutrinos reaching the earth (roughly about half from what it should reach the earth). This is so called the solar neutrino problems and there are several possible solutions. One is that we may not understand the sun very well; therefor we predict the wrong number of neutrinos produced inside the Sun.

This solution can be ruled out easily because we know exactly how much energy is produced by the Sun, and we know how many neutrinos we should expect per unit energy. However a modified version of this explanation is more difficult to reject, as we may be predicting the wrong shape of the neutrino energy spectrum. It turns out that the efficiency of detectors drops down very quickly as we go

down in neutrino energy, so measuring the shape of the neutrino energy spectrum is important.

The second solution may be neutrino oscillations which would indicate that we do not yet know all there is to know about neutrinos. The idea is that an electron-neutrino on its way from the Sun to the Earth can transform itself into e.g. a muon-neutrino and which will escape detection.

Inside the sun, only electron-neutrinos are produced, and we know that all the neutrinos are light particles and are possibly massless. But if they have some small mass, there is a possibility of 'mixing' between them if the so called 'mixing angle' is non-zero. This mixing and oscillations of particles is nothing new in physics; it has been observed in the neutral kaon system. Another explanation is that maybe we don't understand the interactions of neutrinos with the detectors. The trouble is that neutrino interactions are complicated by the fact that most of the neutrons and protons are bound together in oxygen nuclei in water.

4.4 SUPER-KAMIOKANDE EXPERIMENT

The Super-Kamiokande is a water Cerenkov detector for neutrino detection. The experiment is 1000 m underground (2700 m water equivalent) in the Kamioka mine. A large cavity with a dome shape was excavated to use for the construction of the water tank.

The cavity was plastered with 40 - 50 cm thick reinforced concrete on the side wall and the floor, then laid with 3 - 4.5 mm thick stainless steel. The size of the tank is 42 m in height and 39 m in diameter. The total volume is 50,000 m^2; 26 sets of Helmholz coils are

arranged in the tank to eliminate the effect of geo-magnetic field on photomultipliers (PMT). The tank is filled up with pure water and the water is recycles every few days.

The Super-Kamiokande it is consists of two layers of detectors, a inner counter and a outer counter. Support structures for photomultipliers, which divided the tank between inner and outer counter, are built 2.7 m apart from the tank wall; 11,146 20-inch PMT's are placed uniformly over the entire inner surface. The photo-cathode coverage of the PMT's is 40%, which is twice of that of the present Kamiokande. The gap between PMT's is covered with black polyetylene sheets. The total mass of water inside the PMT surface is 32,000 tons.

An additional black polyetylene sheet is set between the inner PMT's to optically isolate between the inner and the outer counter. The outer part, which has 2m in thickness, is used for anti-counters. The extra surface of the outer counter is covered with high-reflection white polyethylen sheets, 1,857 8-inch PMT's are used for the outer counter. At each PMT window, a 60-cm square acyliric plate counting wave-length shifter is equipped to enlarge the light correction and efficiency of the 20-inch Photomultiplier tube.

The inner PMT has a 20-inch diameter and is equipped with a quasihemispherical window with bi-alkali photocathode. The venetian blind type dynodes of 11 stages were chosen to obtain a large collection area for photoelectrons. The bleeder chain board and the cable extraction are surrounded by a waterproof structure consisting of epoxy reign and heat-shrink tube. The structure has been proved to be sufficient for use up to a water depth of 70 m.

The 20-inch PMT was first developed by Hamamatsu photonics Co. for the present Kamiokande. Sufficient performance for a large water Cerenkov detector has been demonstrated during its long run. For Super Kamiokande, several improvements in the dynodes shape, bleeder chain, and etc. enable us to obtain good timing and energy resolution.

A typical gain of the PMT is 10^7 with about 2000 V. The quantum efficiency is typically 20 %. A typical spread of the transit-time is about 3 ns. These performance are enough to carry out experiments in Super Kamiokande.

4.4.1 THE ELECTRONIC SIGNALS

The signal that is created by the swarm of electrons, enters a cable and travels from the photomultiplier to the electronics hut located at the top of the tank. There it enters amplifiers, analog-to-digital converters, (ADC) and other electronics. The signal from all photomultipliers is converted to digital form, analyzed and passed to computers for further analysis.

At many stages decisions have to be made (by electronics or computers) whether there is an interesting physical signal has been seen by the photomultipliers, or whether it is just noise. One very important part of the electronics which makes the inital decision is called the trigger, as it triggers other stages of data processing. This is the way a lot of noise is rejected. After the final computer decision,the data then is stored on magnetic tapes and it is saved to be analyzed later, off-line.

4.4.2 ONLINE DATA ACQUISITION SYSTEM (DAQ)

The data acquisition (DAQ) system for the inner detector of Super Kamiokande collects digitized data from the ATM's (see appendix C) front-end electronics, which are connected to 20-inch photomultipliers. Analog signals are converted to digital values for both integrated charge and timing. The system consists of eight Sun SPARClassic (S4/CL) one for every 20 ATM modules, another S4/CL for data from the trigger module, and one Sun SPARCstation 10 (S4/10) for event reconstruction, basic monitoring and data transfer to the off line computer. There are 4 electronics huts and one central hut.

In the electronics hut, 240 ATM modules (12 TKO boxes) are controlled by 2 S4/CLs. In the central hut, the trigger module is controlled by one S4/CL. The trigger module distributes trigger signals and event counters to 48 GONG (Go/No Go) modules, which distribute those signals to ATM modules.

Data obtained in one TKO box (i.e. 20 ATM modules) are collected by a SMP (Super Memory Partner) module which has internal memory of 2 Mbyte (1 Mbyte per 1 memory) by sparse data scan (SDS). The data of 6 SMP modules are read out by each S4/CL via Bit-3 Sbus-VMEbus interface using block transfer.

The process which reads data from SMP is called 'collector'. Collected data are reconstructed by the process 'sorter' on each S4/CL using the event counter, and the process sender transfers the reconstructed data to the process event builder running on S4/10, which is located in a central hut, via fast network link FDDI. Data sent from 8 S4/CLs are concatenated by this process event builder and

completes event reconstruction. Reconstructed data will be sent to Off - line site every 10 minute whose data size corresponds to about 100 Mbyte.

4.5 DATA REDUCTION

Before serious high energy analysis can be conducted on the data from the Super-K detector, the data must be reduced. For the high energy group this reduction is done by various means, including, raising the trigger level in software, removing or `sparring' through going muon event, as well as removing noise events. This is where the root isolation program were implemented for detecting the good muon events . After the data is received then root isolation program was used by reading each event and then for each event will produces a set of value for each of four dimension.

These values then are feed back in to the original formula for test. Then the real solar neutrinos are detected and there tracks in the tank are find.

4.6 TRACKING

As was discussed previously at each trigger some of the photomultiplier will see some light where the signal created by the light enters a cable and travels from the photomultiplier to the electronics. There it enters amplifiers and analog-to-digital converters. The signal from all photomultipliers is converted to digital form and then they are passed to computers to be saved for further analyzing.

The problem is that we need to ascertain whether the signal produced from PMT's hits is a good event, or is it just some background noise. To determine this, we must find out the actual

coordinates of light in the tank and compare it to some point (zero). In another words, we must track the light. Figure 4.2 shows a solar vertex entering the tank at the X,Y,Z coordinates with a timing of t (initial time).

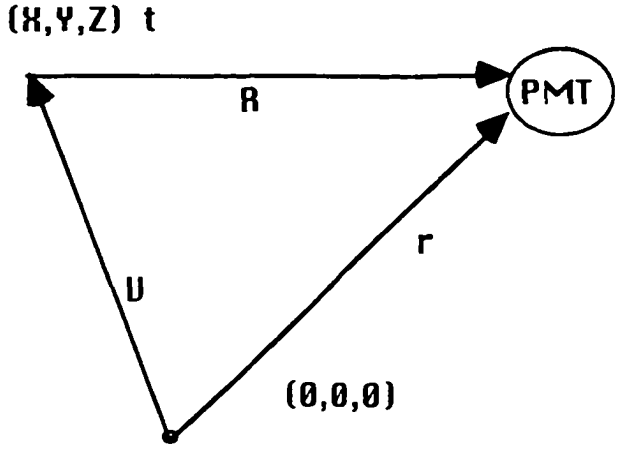


Figure 4.2 Tracking example

If the vertex travails in the direction of R to the PMT, and O is the reference point (zero), then we can say that the distant r is known and also that the measure time is known with respect to point 0. So we can say that

$$\bar{V} + \bar{r} = \bar{R}$$

where

$$\bar{r} = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2 + (z_i - z_0)^2}$$

$$t_i = t_0 + \frac{rng}{c}$$

$$\tau_i = \tau_0 + rng$$

and ng is the index of reflection of pure water, c is the speed of light, t is the measured time, and τ_0, x_0, y_0, z_0 are the unknown coordinates and the entry time. By knowing the above we can derive the function for tracking by subtracting the summation of known distance and time from unknown part .

$$y = \sum_i \frac{(t_i - t_0 - rng)^2}{\sigma_i^2}$$

The σ is the time resolution of the PMT's, and it was assumed the same for all the PMT's. If we take the derivative of above function with respect to each of our four unknowns (see appendix C) then we will end up with four functions and four unknowns which we call X(1), X(2), X(3), and X(4) for t, x, y, and z where this it will enable us to find the track of the vortex.

4.7 RESULT

Each saved data events on computer file have PMT id, timing, and energy. The first and last line of each file shows the beginning and the end of the file with numbers of hits, as shown in Table 4.1.

Table 4.1 shows a sample trigger read out of one event in which 23 photomultipliers were hit. Each photomultiplier had its own id and set of corresponded coordinates to compare to some point in space called zero of the tank in four dimensions (X,Y,X,T).

So each event will have the same numbers and timing of the hit photomultipliers. At this stage decisions have to be made as to whether the signal seen by the photomultipliers is an interesting physical (solar neutrino in this case) or whether it is just noise.

Table 4.1 Event out put for each trigger

+++TUBCAL 23

PMT #	TIME	ENERGY	PMT #	TIME	ENERGY
6168	17.98	1.00	6474	21.83	1.00
6216	11.07	2.00	6520	18.67	1.00
6218	11.84	1.00	6576	23.53	1.00
6264	11.56	1.00	6620	25.68	1.00
6265	13.32	1.00	6623	23.52	1.00
6267	10.46	1.00	6625	25.58	1.00
6313	17.17	1.00	6776	28.61	1.00
6314	13.71	2.00	6782	39.40	1.00
6365	18.43	1.00	6928	42.86	1.00
6372	17.43	2.00	7184	60.14	1.00
6414	19.32	1.00	6467	22.03	1.00
6423	20.00	1.00			

+++EVTEND 3

Root isolation at this point is one very important part of making this decision. A FORTRAN program was developed (see Appendix C) to open each saved file, some of which may have more then 100 event records. Then this program selected the each record on file individually and read in the time and PMT's id from the event file; next the program opened the coordinate file in which it stored coordinate

information for all 13,000 PMT's. This information was read in the sequence of id number from the coordinate file and we matched this data to the record's id where it had been read previously. At this point, the program will create an output file that includes the PMT's id with matching coordinates and timing for each id numbers from which it was read from both input files. A sample of this output file is shown in Table 4.2 (part of table 4.1 is included).

At this point, the root isolation program was modified in order to open the output file created previously by event application. The root isolation program then read in all the information from input files that have all data from each event.

The root isolation program needed to be more user friendly. So it was modified to include more user interfaces, (chapter 2), which means that this program will not need to be edited for new sets of function. An include file was added that needed to be updated for each new set of numbers.

The root isolation program was able to locate successfully the real roots of very complex nonlinear equations; Those results (the out put of root isolation program) were used to enable us find the track of particles. The sample result of root isolation program is shown in Table 4.3. Where X(1) is the time and X(2), X(3), X(4) are respectively x, y, z of the entry coordinates. In case when there are more then one root, these results are put back in the original function in order to choose the smallest root that corresponds to the function value.

Table 4.2 Sample output of SKMAN.FOR

1	6168	522.23	-1607.29	1555.40	539.40	17.98
2	6216	589.08	-1584.01	1343.30	332.10	11.07
3	6218	589.08	-1584.01	1484.70	355.20	11.84
4	6264	654.89	-1557.95	1131.20	346.80	11.56
5	6265	654.89	-1557.95	1201.90	399.60	13.32
6	6267	654.89	-1557.95	1343.30	313.80	10.46
7	6313	719.56	-1529.16	989.80	515.10	17.17
8	6314	719.56	-1529.16	1060.50	411.30	13.71
9	6365	782.96	-1497.69	1060.50	552.90	18.43
10	6372	782.96	-1497.69	1555.40	522.90	17.43
11	6414	844.99	-1463.59	919.10	579.60	19.32
12	6423	844.99	-1463.59	1555.40	600.00	20.00
13	6467	905.54	-1426.92	1060.50	660.90	22.03
14	6474	905.54	-1426.92	1555.40	654.90	21.83
15	6520	964.50	-1387.75	1201.90	560.10	18.67
16	6576	1021.77	-1346.14	1555.40	705.90	23.53
17	6620	1077.24	-1302.17	1060.50	770.40	25.68
18	6623	1077.24	-1302.17	1272.60	705.60	23.52
19	6625	1077.24	-1302.17	1414.00	767.40	25.58
20	6776	1231.95	-1156.89	1272.60	858.30	28.61
21	6782	1231.95	-1156.89	1696.80	1182.00	39.40
22	6928	1367.23	-993.36	1201.90	1285.80	42.86
23	7184	1543.89	-687.39	1272.60	1804.20	60.14

Table 4.3: Sample Root Isolation Output for Super K.

prog: smfrot.f tk=.6 A=-1900 B=1900 tolranc=10**-2

****

* CPU TIME FOR THIS SET OF FUNCTIONS IS 445

****

* NUMBER OF ROOTS ARE= 1

* NUMBER OF NODES ARE= 2

ROOT BOUNDARIES FOUND

X(1) = 0.645268 TO 0.654245

X(2) = 516.9345 TO 518.2567

X(3) = -1365.671 TO -1384.682

X(4) = 1268.9686 TO 1277.9634

ROOT BOUNDARIES FOUND

X(1) = 0.246268 TO 0.288945

X(2) = 772.3345 TO 792.9437

X(3) = -1764.691 TO -1771.342

X(4) = 1286.9435 TO 1291.7234

CHAPTER 5

CONCLUSION

5.1 RESULT

From the previous chapters, we can be reasonably confident about how to isolate roots for nonlinear system methods. The earlier implementation (program) of algorithms was already useful and economical for small systems. With the changes introduced and described in this dissertation, a more reliable and efficient method for finding the root of nonlinear systems now exists.

Following the new parallel algorithm introduced in Chapter 3, the root isolation for nonlinear systems of equation can now handle larger and more complex systems. This new parallel program is more reliable and improved efficiency over sequential program (per each node). The parallel program solves the problem of being computationally intensive time discussed in Chapter 2. This new parallel algorithm (program) can deal with the more complex functions with the large regions and many roots as shown in Figures 5.1, 5.2, 5.3, 5.4. Each of these figures shows that the run time decreases as more nodes are allocated. Figure 5.5 shows the run time for both the sequential and parallel programs. In the parallel case, we can see that any increase in number of function or variables will not have much effect on the run time or CPU time.

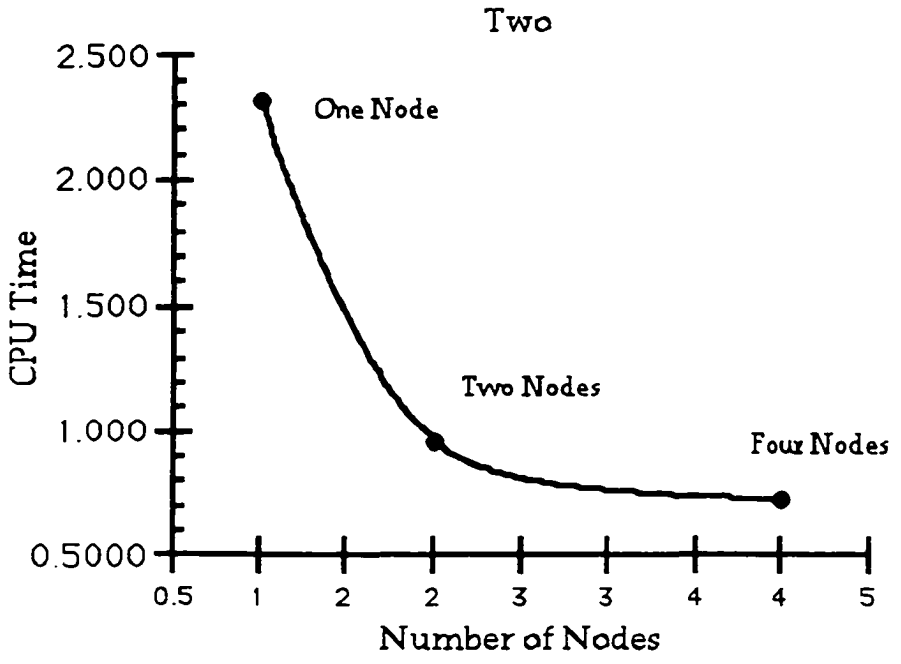


Figure 5.1 CPU time for two function and variable

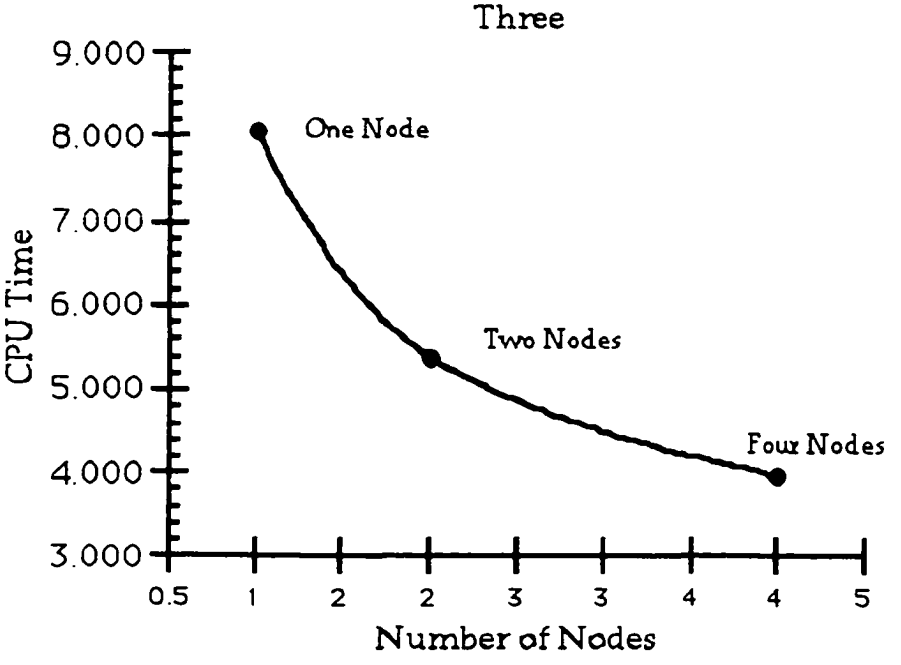


Figure 5.2 CPU time for three function and variable

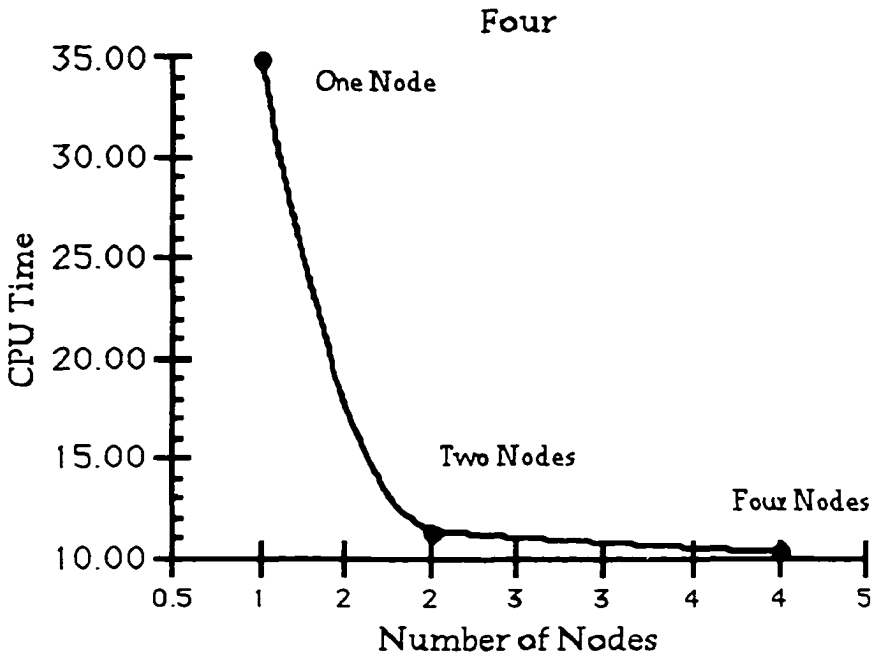


Figure 5.3 CPU time for four function and variable

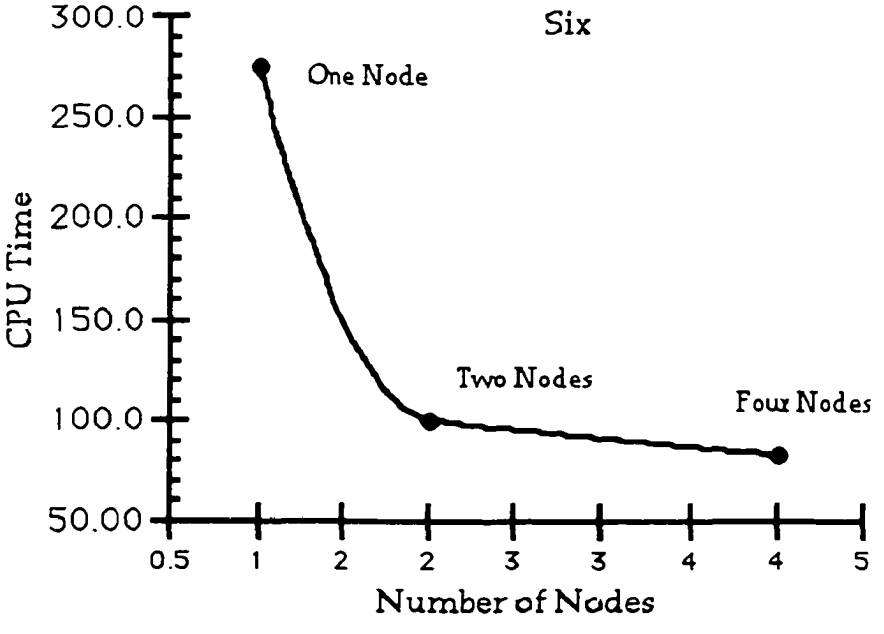


Figure 5.4 CPU time for six function and variable

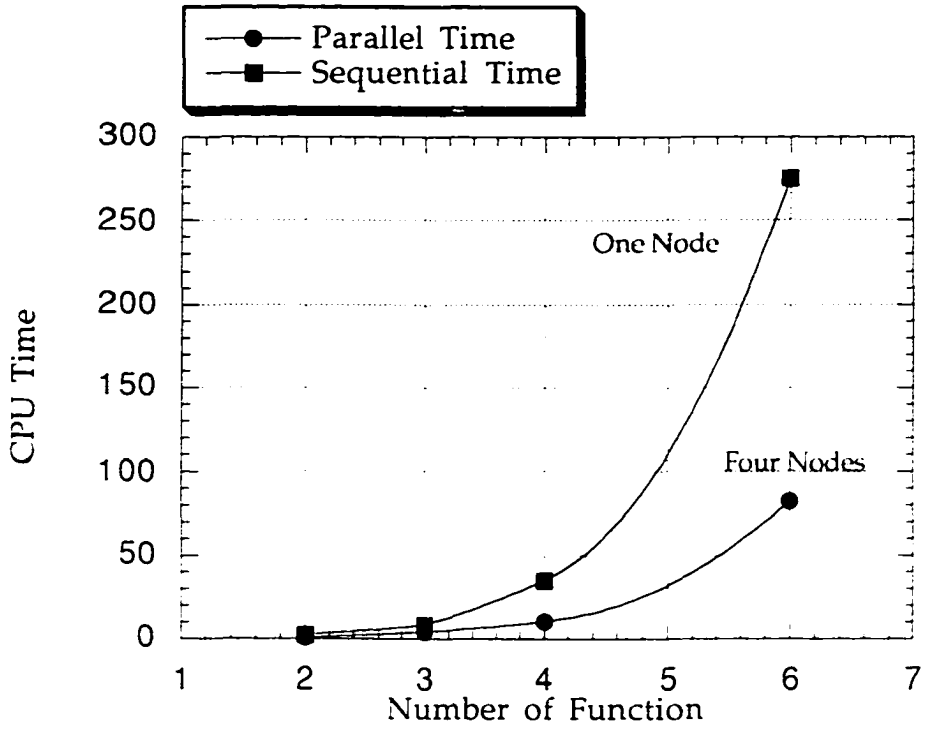


Figure 5.5 CPU time for sequential and Parallel system.

5.2 CONTRIBUTIONS

A computational technique was develop for investigating a metrology of preventing the algorithm structure from converging to the same root along different paths of computation. A parallel version of the program was designed and implemented on MIMD (Multi Instruction Multi Data) architecture. Finally, the metrology presented here has been implemented for a real world application.

One of the more promising contributions of the new program is that it can substitute for a complex system. The new program was

implemented to find the solution to a real complex world application. We were able successfully implement the new program to track the high energy particles (neutrinos) in a super-k experiment in Japan.

As discussed in chapter 4, this program was able to pick the real roots in very complex systems of equations. This result was accomplished off line, and although this was a very important task at some point it should be possible to use this application on line with about 110 Hz. of data. This will help reduce the data before it goes to any storage devices for off line analysis.

5.3 CONCLUDING REMARKS

Root isolation for nonlinear systems of equation is a powerful tool in many disciplines and sub-disciplines. Here, a step has been taken to bridge the gap between theoretical application and implementation of the sequential algorithm developed by Jones and Iyengar.

This parallel algorithm was developed on IPC / 860 MIMD architecture with a limited numbers of nodes. The situation of having more nodes than variables was addressed by this algorithm for small systems of equations, but due to node numbers that was available with IPC860, this has not been tested yet in larger systems.

Now that the parallel algorithm proposed in this research has been proven effective, further research is needed for implement this algorithm on larger Multi-Instruction Multi Data (MIMD) systems where more nodes are available.

This parallel program could be exploited further by implementation on Parallel Virtual Machine (PVM). Point-to-point as

well as multicast data communication could be reliable in various situations so that will make it portable where it is more accessible and also will reduce the cost.

BIBLIOGRAPHY

[Jone 77] B. Jones, " A Heuristic for Root Isolation, " International Journal of Numerical Methods in Engineering 11, 1977.

[Jone 78] B. Jones, W. Wailer and A. Feldman, " Root Isolation Using Function Values, " BIT 18, 1978.

[Jone 84] B. Jones, M. Banerjee and L. Jones, " Root Isolation for Transcendental Equations, " The Computer Journal 27, No. 2, 1984.

[Jone 94] B. Jones and S. Iyengar, " Approximate Root Isolation for Nonlinear Systems by Monte-Carlo, " Computers and Mathematics with Applications 27, No. 7, 1994.

[Rals 60] Ralston, A. and Wilf, H. S. " Mathematical Methods for Digital Computers, " New York, Wiley, p. 14, 1960.

[Cuyt 87] Cuyt, A., and Wuytack, L. " Nonlinear Methods in Numerical Analysis," Amsterdam, North Holland, Chapter 2, 1987.

[Grav 79] Graves-Morris, P.R. " Approximation and Its Applications, " Lecture Notes in Mathematics, vol. 765, L. Wuytack, ad. Berlin, Springer-Vedag, 1, 1979.

[Stoer 80] Stoer, J., and Bulirsch, R. " Introduction to Numerical Analysis, " New York, Springer-Verlag, Chapter 5, 1980.

[Acto 90] Acton, F. S. " Numerical Methods That Work, " corrected edition, Washington, Mathematical Association of America, Chapters 2, 7, and 14, 1990.

[Rals 78] Ralston, A., and Rabinowitz, P. " A First Course in Numerical Analysis, " 2nd. ad. New York, McGraw-Hill, Chapter 8, 1978.

[Houe 70] Householder, A. S. " The Numerical Treatment of a Single Nonlinear Equation, " New York, McGraw-Hill, 1970

[Radk 90] C. E. Radke, " The use of Quadratic Residue Research, " ACM Communications 13 , 2, pp. 103 -105, February 1990.

[Davi 88] Davies M. and, Dawson B. "an Automatic Search Procedure for Finding Real Zeros, " Numerische mathematik 31, pp. 299 - 312, 1988.

[Grop 94] W. Gropp, E. Lusk, and A.Skjellum, " Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, 1994.

[Atha 88] W. C. Athas, and C. L. Seitz, "Multicomputers: Message-passing concurrent computers, "IEEE Compute., pp. 9-25, Aug. 1988.

[Foun 90] T. J. Fountain, and M. J. Shute," Parallel Computing, " Holland Press, 1990.

[Hell 78] D. Heller, "A Survey of Parallel Algorithm in Numerical Algebra," SIAM Rev., vol. 20, pp. 740-777, Oct. 1978.

[Leut 93] S. Leutenegger and X.-H. Sun, " Distributed Computing Feasibility in a Non-Dedicated Homeneous Distributed System, " In Proceedings of Supercomputing, pp. 143-152, 1993.

[Begu 95] A. Beguelin, B Lowekamp, E. Seligman, "Dome: Parallel Programming in a Multi - User Environment, " Tech. Rep., Technical paper submission for Supercomputing, 1995.

[Shoo 83] Shooman, Martin l., "Software Engineering: Design, Reliability, and Management, " New York, McGraw-Hill, 1983.

[Ohno 82] Ohno, Y., " Requirements Engineering Environments, " New York, North Holland 1982.

[Boot 83] Booth Grayce M., " The Design of Complex Information System, " New York, McGraw-Hill, 1983.

[Joer 85] Joerger, F. A., " Functional Instrumentation Modules, " IEEE Transactions on Industry Applications, vol. IA-11, Nov. 1985.

[Shel 92] Frederick T. Sheldon, " Reliability Measurement, ". IEEE Software, vol. 0740/7459, July 1992.

[Good 91] Michael T. Goodrich. " Using Approximation Algorithms to Design Parallel Algorithms, " In Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science, pp. 711-722, 1991.

[Ston 75] H. S. Stone, " Parallel Tridiagonal Equation Solvers, " ACM Trans. on Mathematical Software, 1(4) pp. 289-307, 1975.

[Fost 94] I. Foster, B. Avalani, and M. Xu, " A Compilation System that Integrates High Performance Fortran and Fortran, " Proc. 1994 Scalable High Performance Computing Conf., IEEE, 1994.

[Hatc 91] P. Hatcher and M. Quinn, " Data-Parallel Programming on MIMD Computers," MIT Press, 1991.

[Panc 90] C. Pancake and D. Bergmark, " Do Parallel Languages Respond to the Needs of Scientific Programmers?, " Computer 23 (12), 13-23, 1990.

[Gust 88] J. Gustafsopn, G. Montry, and R. Benner, " Development of Parallel Methods for a Hypercube," SIAM journal on Scientific and Statistical Computing, 1988.

[Cyph 93] R. Cypher, A. Ho, S. Konstantinidou and p. Messina, " Architectural Requirements of Parallel Scientific Applications with Explicit Communication, " In 20th Annual International Symposium on Computer Architecture, May 1993.

[Zaho 93] C. McCann, and J. Zahorjan. " A Dynamic Processor Allocation Policy for Multi Programming, " In ACM Transactions on Computer System, volume 11, pp. 146-178, May 1993.

[McCa 94] C. McCann, and J. Zahorjan. " Processor Allocation Policies for Message Passing Parallel Computers, " In proceedings of the 1994 ACM Conference on Measurement and Modeling of Computer Systems, pp. 19-32, Feb. 1994.

[Quin 94] Michael J. Quinn, " Parallel Computing, " McGraw-Hill, Inc. 1994.

[Bruc 93] Lester P. Bruce, " The Art of Parallel Programming, " Prentice-Hall, Inc. 1993.

[Geal 94] Curtis F. Gerald, " Applied Numerical Analysis," Addison- Wesaly Publishing Company, 1994.

[Corm 90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, " Introduction to Algorithms, " MIT Press, McGraw-Hill Inc. 1990.

[Hwan 93] Kai Hwang, " Advanced Computer Architecture, " McGraw-Hill Inc. 1993.

[Jaja 93] Joseph Jaja, " Introduction to Parallel Algorithms," Addison-Wesaly Publishing Company, 1993.

[Wolf 89] L. Wolfenstein, " Solar Neutrinos, " In 13th Annual International Symposium on Neutrino physics and Astrophysics, pp. 102-114, June 1989.

[Morr 94] D. Morrison, " The Steady Vanishing of Solar Neutrino Problem," In 16th Annual International Symposium on Neutrino physics and Astrophysics, pp. 276 - 284, June 1994.

[Davi 90] R. Davis, "Neutrinos from the Sun," In 21 st Annual International Symposium on Cosmic Ray physics and Astrophysics, pp. 129 - 141, January, 1990.

[Land 90] K. Lande, and C. K. Lee, " Time Dependence of the Solar Neutrino Flux, " In 16th Annual International Symposium on Cosmic Ray physics and Astrophysics, pp. 143 - 152, January, 1990.

[Perk 89] Donald H. Perkins, " Introduction to High Energy Physics, " Addison-Wesley Inc., 1989.

[Bark 83] W. H. Barkas, "Nuclear Research Emulsions," New York Academic Press, 1983.

[Beck 90] R. Becker, R. Svoboda, and H. W. Sobel, " A Search for Muon Neutrino Oscillations with the IMB Detector," In 21 st. Annual International Symposium on Cosmic Ray Physics and Astrophysics, January, 1990.

[Iyeng 86] S. S. Iyengar, "Parallel Algorithms for a Class of Computational Problems," Advance in Computer Academic Press, 1986.

APPENDIX A

PARALLEL PROGRAM

List of Argument

NOTE

To understand this application, the following is the list of arguments, functions, and symbols it were used. This should help to read and understand the logic that was used for designing this parallel application.

PARALLEL SUBROUTINS

NBYTES =NUMBER OF BYTES PER MESSAGE
NODES = NUMBER OF NODES
IAM=NODE NAME
NODID = NODE ID NUMBER
HOSTNID MYHOST() HOST NODE ID
HOSTPID 1 HOST PROCESS ID
INITTYPE 10 TYPE OF INITIAL MESSAGE
NODETYPE 20 TYPE OF NODE MESSAGES
TIMETYPE 30 TYPE OF TIME MESSAGE (SENT TO HOST)
COUNTTYPE 40 TYPE OF COUNT MESSAGE (SENT TO HOST)
INITSIZE 8 SIZE OF INITIAL MESSAGE (IN BYTES)
TIMESIZE 4 SIZE OF TIME MESSAGE (IN BYTES)
COUNTSIZE 4 SIZE OF COUNT MESSAGE (IN BYTES)
MAXMSGSIZE 65536 MAX. MESSAGE SIZE ALLOWED (IN BYTES)
COUNT, TEMPORARY STORAGE FOR COUNTER
MSGLEN, LENGTH OF MESSAGE (IN BYTES)
MSGBUFF, MESSAGE BUFFER
MENODE, NODE ID RETURNED BY MYNODE()
MEPID, PROCESS ID RETURNED BY MYPID()

RCNT,	NUMBER OF BYTES IN INCOMMING MESSAGE
RNODE,	NODE OF ORIGIN OF INCOMING MESSAGE
RPID	PID OF ORIGIN OF INCOMING MESSAGE

SUBROUTINE ROOT

A ARRAY = THE LOWER BOUND FOR EACH COORDINATE.
B ARRAY = THE UPPER BOUND FOR EACH COORDINATE.
M = THE NUMBER OF EQUATIONS (2 IN THIS CASE)
N = THE NUMBER OF COORDINATES (3 IN THIS CASE)
TK = THE (MEAN SQUARED)/(VARIANCE) TEST PARAMETER.
E = THE TERMINAL INTERVAL TOLERANCE SMALLER THAN THIS, IT IS CONSIDERED TO CONTAIN A ROOT.
OUTSTK = ARRAY WHICH CONTAINS THOSE REGIONS WHICH HAVE ROOTS.
IAL = AN INDICATOR FOR INDICATING WKSTK OVERFLOW. IAL IS SET TO ZERO IF THE INITIAL VALUE WAS TOO SMALL. IAL IS DIMENSION WKSTK
NROOT = NROOT IS AN INTEGER . ON EXIT NROOT CONTAINS THE NUMBER OF ROOTS WHICH HAVE BEEN FOUND IN THE WHOLE REGION

EPS = THE CONSOLIDATION PARAMETER
XMEAN1 = OLD MEAN
XMEAN2 = NEW MEAN
VAR2 = NEW VARIANCE
XMENS1 = OLD MEAN SQUARED
XMENS2 = NEW MEAN SQUARED
WKSTK = WORKING SPACE AREA
FF = FUNCTION VALUE
X = COORDINATE VALUES.
EPS = THE (MEAN SQUARED)/(VARIAMCE) CONVERGENCE

CRITERION
LNXT = LIST NUMBER
NX = THE NUMBER OF ENTRIES IN FREE LIST
NZ = NUMBER OF LISTS
KN = THE CURRENT TOTAL NUMBER OF POINTS ON EACH DIMENSION
IFB = THE FLAG FOR GOING FORWARD OR BACKWARD IN LIST PROCESSING (INDICATES WHETHER WE ARE DEALING WITH AN OLD LIST OR A NEW LIST
IENTRY = AN INDICATOR USED TO CHECK WHETHER THE OUTSTK IS EMPTY OR NOT (IENTRY IS INCREMENTED AS REGIONS ARE ADDED TO THEOUTSTACK)
ID = THE DIMENSION WHICH IS PARTITIONED CURRENTLY (I.E., WHICH OF THE INTERVALS (A(I),B(I)) IS BEING CUT)
LEVEL = AN INDICATOR USED TO INDICATE WHICH DIMENSION SHOULD BE PARTITIONED NEXT TIME
NE = THE NUMBER OF ENTRIES OF OUTSTK (ON EXIT, NUMBER OF ROOTS IS SET TO NE)
INDEX = THE NUMBER OF REGIONS WE HAVE ON WKSTK LNXT INDICATES THE 0TH LIST BECAUSE WE HAVE NOT CREATED ANY YET, NE = NO ENTRIES ON OUTSTACK, NX AND NZ PROVIDE FOR 2000 ELEMENTS PER LIST AND 70 LISTS, RESPECTIVELY
C,X,A,B = ARE OF DIMENSION N.
XMEAN1,XMEAN2,VAR2,XMENS1,XMENS2,SUM,SUM2,AND FF ARE OF DIMENSION M.
COMPUTATION DIMENSION FOR OUTSTK WKSTK XF AND LPT ARE DESCRIBED IN DOCUMENT.

SUBROUTINE POINT

THE ARRAY XF FURNISHES LIST ELEMENTS
XF(.,1) = CONTAINS THE LINK

XF(.,2 TO N1+1) = CONTAINS COORDINATES
XF(.,N1+1 TO N2+N1+1) = CONTAINS FUNCTION VALUES
LPT(I) = POINTS TO THE ITH LIST (HEADER)
N1 = THE NUMBER OF COORDINATES
N2 = THE NUMBER OF FUNCTIONS ()
L = THE NUMBER OF HEADERS
N = THE NUMBER OF LIST ELEMENTS
F = THE ARRAY OF FUNCTION VALUES
X = THE ARRAY OF COORDINATES
IFB = INDICATES CREATION OF A NEW LIST (-1),
ADDITIONS TO A NEW LIST(1),FREEING OF A LIST(-2),
OR USE OF AN OLD LIST (2)

The parallel Program for root isolation of system of nonlinear equation in its final form, is listed in the following pages.

```
      PROGRAM FOR ROOT.
C     ARGUMENTS AND PARAMETERS ARE DESCRIBED IN
C     ROOT AND POINT.
      OPTIONS/CHECK=NOOVERFLOW
      INCLUDE 'FCUBE.H'
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DOUBLE PRECISION  T1, T2, T3
      DIMENSION A(10),B(10),OUTSTK(100,2),DIFF(10,2),JJJ(10),
      DIMENSION  GLO_OUTSTK(400,2), LENS(100,2)
      INTEGER NCNT, NODES
C     SET ARGUMENTS AND INPUT PARAMETERS FOR MAIN.
      TYPE*,'TYPE IN HOW MANY FUNCTIONS YOU HAVE (EG
      "3"):  '
      READ(5,*)M
      TYPE*,'TYPE IN HAW MANY UNKNOWN YOU HAVE (EG
      "3"):  '
      READ(5,*)N
      TYPE*,'LOWER BOUND IS -10 AND UPER BOUND IS 10'
      DATA TK/6./,E/1.E-2/
      DATA N/3/,M/3/
      DATA A/-10.,-10.,-10.,-10.,-10.,-10.,-10.,-10.,-10.,-10./
```

```
      DATA B/10.,10.,10.,10.,10.,10.,10.,10.,10./
      OPEN(UNIT=6,FILE='ROOT.OUT',STATUS='UNKNOWN',RE
      CL=102       -     $ 4)
      T1 =DCLOCK()
       IAL=2300
       KK=0
      JJ=0
      JJJ(KK)=0
C
C****ISOLATE  ROOTS.
C
      CALL ROOT(A,B,M,N,TK,E,OUTSTK,IAL,NROOT)
C
C****GET RESULT FROM EACH NODE
C
       NBYTES=4*NROOT
      NODES=NUMNODES()
      CALL GCOL(NBYTES,4,LENS,4*NODES,NCNT)
       CALL GCOLX(OUTSTK(1,1),LENS,GLO_OUTSTK(1,1))
      CALL GCOLX(OUTSTK(1,2),LENS,GLO_OUTSTK(1,2))
C
C**** CHECK FOR WORKSTACK OVERFLOW
C
      IF(IAL.NE.0)          GO TO 200
      WRITE(6,100)
  100   FORMAT(5X,'WKSTK IS OVERFLOW')
      GO TO 800
C
C**** CHECK FOR NUMBERS OF ROOTS
C
   200 IF (NROOT.EQ.0)       GO TO 750
      IF (NROOT.GT.1)       GO TO 210
      IF (NROOT.EQ.1)       GO TO 290
C
C**** CHECK FOR ROOT IN THE SAME REG.
```

```
  210 KK=0
      DO 220 I1=1,NROOT-1
      DO 230 I2=(I1+1),NROOT
      DO 240 J=1,N
      KK=KK+1
      DIFF(KK,1)=ABS((OUTSTK(J+N*(I1-1),1))
     $          -(OUTSTK(J+N*(I2-1),1)))
       DIFF(KK,2)=ABS((OUTSTK(J+N*(I1-1),2))
     $          -(OUTSTK(J+N*(I2-1),2)))
  240 CONTINUE
  230 CONTINUE
  220 CONTINUE
C
C**** SET FLAG FOR SAME ROOTS
C
      DO 250 KK=1,N
      DO 260 I=1,2
      IF (DIFF(KK,I).GT.(1.E-2)) GO TO 290
      IF (DIFF(KK,I).LE.(1.E-2)) JJJ(KK)=1
      IF (JJJ(KK).EQ.1) NROOT=NROOT-JJJ(KK)
  260 CONTINUE
  250 CONTINUE
      IF (JJJ(KK).EQ.1) THEN
      NROOT=NROOT-JJJ(KK)
      ENDIF
  290 T2=DCLOCK()
      T3=T2-T1
      CPUTIME=T3
      PRINT*, '**********RESULT IN ROOT.OUT**********'
C
C**** PROCEED TO WRITE OUT ANY ROOTS.
C
      PRINT *,'        ****************************************'
      PRINT *,'    * CPU TIME FOR THIS SET OF FUNCTI.
                  IS',CPUTIME
```

```
      PRINT  *,'        ****************************************'
      PRINT *,'     * NUMBER OF ROOTS ARE=' ,NROOT
      PRINT  *, '       ****************************************'
      WRITE(6,291),CPUTIME,NROOT
291   FORMAT(10X,'CPU TIME IS',2X,D15.9,5X,'NUMBER OF
           ROOTS ARE',I2,/)
      DO 600 I=1,NROOT
      NLINE=NROOT*2/2
      WRITE(6,300)
 300  FORMAT(14X,'ROOT BOUNDARIES FOUND',/)
      DO 500 J=1,N
      JJ=JJ+1
      WRITE(6,400)J,OUTSTK(JJ,1),OUTSTK(JJ,2)
 400  FORMAT(5X,'X(',I2,') =',F12.6,3X,'TO',F12.6,/)
 500  CONTINUE
 600  CONTINUE
      GO TO 800
 750  WRITE(6,760)
 760  FORMAT(//5X,'NO ROOTS')
 800  STOP
      END
C
      SUBROUTINE ROOT(A,B,M,N,TK,E,OUTSTK,IAL,NROOT)
C
C**** INITIALIZATION, DECLARATIONS
C
      DOUBLE PRECISION SUM,SUM2,TEMSUM,TEMSUM2
      DIMENSION X(10),XMEAN1(10),XMEAN2(10),VAR2(10)
     $,XMENS1(10),XMENS2(10),A(10),B(10),OUTSTK(100,2),
     $WKSTK(2300,3),SUM(10),SUM2(10),FF(10),TEMSUM2(10)
      DIMENSION XF(2000,21),LENS(2000,21),GXF(4000,21)
     $,LPT(70),TEMSUM(10)
      COMMON /SAM/ LFREE,LNXT,LPT,XF,LF
      INTEGER NODES,IAM,LEN(16),ALEN
      DATA EPS1/.5/
```

```
      LNXT=0
      NE=0
      EPS=TK*E
      NX=2000
      NZ=70
      IND=0
      IAM=MYNODE()
      NODES=NUMNODES()
      NPID=MYPID()
C
C**** TO CREATE FREE LIST; NOW WE WORK WITH THE 1ST LIST
C
      LFREE=1
      M2=NX-1
C
C**** SET THE POINTER IN THE FIRST COLUMN OF XF; FLOATS
C     ELEMENTS OF LISTS
C
      DO 10 I=1,M2
   10 XF(I,1)=FLOAT(I+1)
C
C**** XF(NX,1)=0. INDICATES THE END OF LIST
C
      XF(NX,1)=0.
C
C**** CREATE HEADER FOR LIST
C
      LNXT=LNXT+1
      LPT(LNXT)=0
      IFB=2
      KK=0
      NN=-1
      IFLAG=0
      IENTRY=0
      NROOT=0
```

```
C**** INITIAL NUMBER POINTS FOR COMPUTING MEAN AND
C      VARIANCE (THESE ARE FEDINTO SUBROUTINE SYSTEM)
C
   20  K=10
       KN=K
C
C**** SET OLD MEAN INITIALLY
C
       DO 30 I=1,M
       XMENS1(I)=999.
   30  CONTINUE
C
C**** INITIALLY ZERO OUT SUM AND SUM2
C
       DO 40 I=1,M
       SUM(I)=0.
       SUM2(I)=0.
   40  CONTINUE
       KXX=0
       GO TO 70
C
C**** IF NOT ALL OF (MEAN SQUARED)/(VARIANCE)
C      CONVERGED, THEN GENERATE MORE POINTS IN THE
C      REGION
C
   50  DO 52 I=1,M
       IF(IAM.NE.0) SUM(I)=0
       IF(IAM.NE.0) SUM2(I)=0
   52  CONTINUE
       K=KN
       KN=KN+KN
       DO 60 I=1,M
       XMENS1(I)=XMENS2(I)
   60  CONTINUE
   70  IF(IFB.NE.-1) GO TO 71
```

```
      IF(NODES.EQ.1) GO TO 71
      ALEN=8000*7
      DO 43 I=1,16
   43 LEN(I)=ALEN
      CALL GCOLX(XF,LEN,XF)
      DO 44 J=1,7
      DO 45 I=1,2000
      GXF(2*I-1,J)=XF(I,J)
      GXF(2*I,J)=XF(I,J+7)
   45 CONTINUE
   44 CONTINUE
      DO 48 I=1,2000
      GXF(I,1)=I-1
   48 CONTINUE
      DO 46 J=1,7
      DO 47 I=1,20
      XF(I,J)=GXF(I,J)
   47 CONTINUE
   46 CONTINUE
      DO 49 I=1,70
   49 LPT(I)=LPT(I)+LPT(I)
      IF(IAM.NE.0)GO TO 91
      NODES=1
      K=K*2
   71 NPN=K/NODES
      IMIN=(NPN*IAM)+1
      IMAX=IMIN+(NPN-1)
      IF(IAM.EQ.NODES-1)IMAX=K
      DO 90 I=IMIN,IMAX
   73 CALL POINTS(B,A,FF,NX,NZ,N,M,IFB)
      DO 80 L=1,M
      SUM(L)=SUM(L)+FF(L)
      SUM2(L)=SUM2(L)+FF(L)*FF(L)
   80 CONTINUE
   90 CONTINUE
```

```
   91  CALL GSYNC()
       CALL GDSUM(SUM,M,TEMSUM)
       CALL GDSUM(SUM2,M,TEMSUM2)
       DO 100 I=1,M
       XMEAN2(I)=SUM(I)/FLOAT(KN)
       XMENS2(I)=XMEAN2(I)**2
       TEMP=(SUM(I)**2)/FLOAT(KN)
       VAR2(I)=(SUM2(I)-TEMP)/FLOAT(KN-1)
  100  CONTINUE
C
C**** TEST IF (MEAN SQUARED)/(VARIANCE) CONVERGED
C
       DO 110 I=1,M
   110 XMENS2(I)=XMENS2(I)/VAR2(I)
       DO 120 I=1,M
       RATIO=XMENS2(I)
       DIFF=RATIO-XMENS1(I)
       IF(ABS(RATIO).LE.EPS1) RATIO=1.0
       IF(ABS(DIFF).GT.EPS1*ABS(RATIO)) GO TO 50
   120 CONTINUE
C
C**** TEST EACH EQUATION INDEPENDENTLY FOR ZERO IN R
C
       DO 130 I=1,M
       IF(ABS(XMENS2(I)).GT.TK)   GO TO 280
   130 CONTINUE
       IFB=-1
       IF(IFLAG.NE.0)          GO TO 300
C
C**** PARTITION REGION
C
       ID=1
       LEVEL=2
C
C**** BISECT THE ID-TH DIMENSION
```

```
      BPRIME=(B(ID)+A(ID))/2.
      INDEX=1
C**** AFTER PARTITIONING A REGION, PUSH THOSE TWO
      SMALL REGIONS ONTO WKSTK
      WKSTK(1,1)=A(1)
      WKSTK(1,2)=BPRIME
      WKSTK(1,3)=FLOAT(LEVEL)
      DO 140 I=2,N
      WKSTK(I,1)=A(I)
      WKSTK(I,2)=B(I)
      WKSTK(I,3)=FLOAT(LEVEL)
  140 CONTINUE
      IA=N+1
      IB=N+N
      IAA=IA+1
      INDEX=INDEX+1
      IC=IA-N
      WKSTK(IA,1)=BPRIME
      WKSTK(IA,2)=B(IC)
      WKSTK(IA,3)=FLOAT(LEVEL)
      DO 150 I=IAA,IB
      IC=I-N
      WKSTK(I,1)=A(IC)
      WKSTK(I,2)=B(IC)
      WKSTK(I,3)=FLOAT(LEVEL)
  150 CONTINUE
      IFLAG=1
C
C**** POP THE WKSTK AND GET A REGION TO BE PARTITIONED
C
  160 DO 170 I=1,N
      IE=(INDEX-1)*N+I
      A(I)=WKSTK(IE,1)
      B(I)=WKSTK(IE,2)
      WK=WKSTK(IE,3)
```

```
      ID=IFIX(WK)
  170 CONTINUE
      IF (IFB.EQ.-2)          GO TO 20
      IF(IND.EQ.1)            GO TO 190
      IF(IAM.NE.MOD(INDEX,4)) GO TO 20
C
C**** CHECK IF THE SIZE OF THE REGION IS SMALL ENOUGH TO
C     BE PUT ON OUTSTK
      DO 180 I=1,N
      BI=B(I)
      IF(ABS(BI).LE.E) BI=1.0
      IF(ABS(B(I)-A(I)).GT.E*ABS(BI))  GO TO 20
  180 CONTINUE
C
C**** PUT REGION IN OUTSTACK
C
  190 IF(IND.EQ.1) IFB=-2
      IND=IND+1
      IF(IND.EQ.2) IND=0
      NN=NN+1
      KK=KK+1
      NE=KK*N
      NNN=NN*N+1
      DO 200 I=NNN,NE
      IZ=I-NN*N
      OUTSTK(I,1)=A(IZ)
      OUTSTK(I,2)=B(IZ)
  200 CONTINUE
C
C**** CONSOLIDATE ADJOINING INTERVALS AND CHECK TO SEE
C     IF OUTSTK IS EMPTY
      IF(IENTRY.EQ.0)         GO TO 250
  210 DO 220 I=NNN,NE
      IMN=I-N
      OUTST1=OUTSTK(I,1)
```

```
      OUTST2=OUTSTK(I,2)
      IF(ABS(OUTSTK(I,1)).LE.EPS)OUTST1=1.0
      IF(ABS(OUTSTK(I,2)).LE.EPS)OUTST2=1.0
      IF(ABS(OUTSTK(IMN,1)-
      OUTSTK(I,1)).GT.ABS(OUTST1)*EPS.AND.
     $ ABS(OUTSTK(IMN,1)-
     $ OUTSTK(I,2)).GT.ABS(OUTST2)*EPS.AND.
     $ ABS(OUTSTK(IMN,2)OUTSTK(I,2)).GT.ABS
     $ (OUTST2)*EPS.AND.
     $ ABS(OUTSTK(IMN,2)-OUTSTK(I,1)).GT.ABS(OUTST1)*EPS)
     $ GO TO 250
  220 CONTINUE
  230 DO 240 I=NNN,NE
      IMN=I-N
      OUTSTK(IMN,1)=AMIN1(OUTSTK(IMN,1),OUTSTK(I,1))
      OUTSTK(IMN,2)=AMAX1(OUTSTK(IMN,2),OUTSTK(I,2))
  240 CONTINUE
C**** SINCE TWO REGIONS ARE CONSOLIDATED INTO
C     ONE,DELETE ONE
C**** REGION'S SPACE FROM OUTSTK
      NN=NN-1
      KK=KK-1
      NE=NE-N
      NST=NE-N+1
      IF(NST.EQ.1)          GO TO 270
      NNN=NST
      GO TO 210
  250 IENTRY=IENTRY+1
      IF(IENTRY.EQ.1)       GO TO 270
      DO 260 I=NNN,NE
      IMN=I-N
      IF((OUTSTK(IMN,1).GT.OUTSTK(I,1).OR.
     $ OUTSTK(IMN,2).LT.OUTSTK(I,2)).AND.(
     $ OUTSTK(IMN,1).LT.OUTSTK(I,1).OR.
     $ OUTSTK(IMN,2).GT.OUTSTK(I,2)))
```

```
     $ GO TO 270
  260  CONTINUE
       GO TO 230
  270  INDEX=INDEX-1
       IF(INDEX.NE.0)          GO TO 160
       NROOT=NE/N
       RETURN
  280  IFB=-2
       IF(IFLAG.NE.0)          GO TO 270
       NROOT=NE
       RETURN
C
C**** CHECK TO SEE IF WKSTK OVERFLOW
C
  300  IF((INDEX*N+N).GT.IAL) IAL=0
       IF(IAL.EQ.0)RETURN
       LEVEL=ID+1
C
C**** AFTER SUBDIVIDING XN, CYCLE BACK AND SUBDIVIDE IN
C      THE INTERVAL OF X1
       IF(LEVEL.GT.N) LEVEL=1
       BPRIME=(B(ID)+A(ID))/2.
       IG=(INDEX-1)*N+ID
       WKSTK(IG,2)=BPRIME
       DO 310 I=1,N
       ITEMP=(INDEX-1)*N+I
       WKSTK(ITEMP,3)=FLOAT(LEVEL)
  310  CONTINUE
       INDEX=INDEX+1
       IH=(INDEX-1)*N+1
       IJ=(INDEX-1)*N+N
       DO 330 I=IH,IJ
       IK=I-(INDEX-1)*N
       IL=I-N
       IF(IK.EQ.ID)            GO TO 320
```

```
      WKSTK(I,1)=WKSTK(IL,1)
      WKSTK(I,2)=WKSTK(IL,2)
      WKSTK(I,3)=WKSTK(IL,3)
      GO TO 330
  320 WKSTK(I,1)=BPRIME
      WKSTK(I,2)=B(ID)
      WKSTK(I,3)=FLOAT(LEVEL)
  330 CONTINUE
      GO TO 160
      END
**********************************************************************
C     THIS ROUTINE MANIPULATES LISTS, STORES AND
C     RETRIEVES POINTS, NEW LISTS ARE CREATED TO STORE
C     POINTS IN NEW REGIONS,  OLD LISTS ARE USED TO
C     FURNISH POINTS FOR THE CURRENT REGION.
C
      SUBROUTINE POINTS(B,A,F,L,N,N1,N2,IFB)
      DIMENSION B(10),A(10),F(10),XF(2000,21),LPT(70)
     $,X(10),LENS1(4000,42),LENS2(4000,42),LENS3(4000,42)
     $,GXF(2000,21)
      COMMON /SAM/ LFREE,LNXT,LPT,XF,LF
      INTEGER NODES,NCNT,LEN(16),ALEN, IAM
      DATA LAST/0/
      DATA SK/0/
      DATA NOLD/10317/
C
C*DETERMINE IF CREATE NEW LIST OR USE OR DELETE OLD LIST
C
      IAM=MYNODE()
      NODES=NUMNODES()
      IF(IFB.EQ.-2)          GO TO 190
      IF(LNXT.LE.L)            GO TO 10
      IF(IFB.EQ.-1) LNXT=LNXT+1
      IFB=1
      GO TO 110
```

```
 10     DDD=1
        IF(IFB.GT.0)          GO TO 30
        IFB=1
        LAST=LPT(LNXT)
C
C**** CREATE A NEW LIST (NEW REGION EXISTS)
C
        LNXT=LNXT+1
        IF(LNXT.GT.L)         GO TO 20
        LPT(LNXT)=0
   20   DDD=2
        IF(LNXT.GT.L)         GO TO 110
        LAST2=0
   30   DDD=3
        IF(IFB.EQ.2)          GO TO 220
   40   DDD=4
        IF(LAST.EQ.0)         GO TO 110
C
C**** IS LIST POINT IN CURRENT REGION?
C
        DO 50 I=1,N1
        XFF=XF(LAST,I+1)
        IF(A(I).GT.XFF.OR.B(I).LT.XFF)GO TO 100
   50   CONTINUE
C
C**** ASSIGN A POINT FROM THE LIST
C
        DO 60 I=1,N2
   60   F(I)=XF(LAST,N1+1+I)
        XYZ=XF(LAST,1)
        LAST=IFIX(XYZ)
C
C**** DELETE FROM PAST LIST, ADD TO CURRENT LIST
C
        IF(LAST2.NE.0)        GO TO 70
```

```
C**** LAST2=0 MEANS WE WANT TO DELETE THE TOP ELEMENT
C
      LD2=LPT(LNXT-1)
      XZZ=XF(LD2,1)
      LPT(LNXT-1)=IFIX(XZZ)
      GO TO 80
C
C**** LD2 POINTS TO THE ELEMENT WHICH WE WANT TO
C     REMOVE
   70 DDD=7
      XYY=XF(LAST2,1)
      LD2=IFIX(XYY)
C
C**** AFTER REMOVING AN ELEMENT, LINK THE LIST
C
      XF(LAST2,1)=XF(LD2,1)
   80 DDD=8
      IF(LPT(LNXT).EQ.0)      GO TO 90
      LN=LPT(LNXT)
      LPT(LNXT)=LD2
C
C**** AFTER ADDING A NEW ELEMENT, LINK THE LIST
C
      XF(LD2,1)=FLOAT(LN)
      RETURN
C
C**** IF THE LIST WHICH WE MOVE AN ELEMENT TO WAS AN
C     EMPTY LIST, THEN
   90 DDD=9
      LPT(LNXT)=LD2
      XF(LD2,1)=0.
      RETURN
  100 DDD=10
      LAST2=LAST
      XYZ=XF(LAST,1)
```

```
      LAST=IFIX(XYZ)
      GO TO 40
C
C**** EVALUATE SYSTEM OF EQUATIONS
C
  110 CALL SYSTEM(A,B,N1,F,X,IFB)
      IF(LFREE.EQ.0.OR.LNXT.GT.L) RETURN
C
C**** ADD NEW POINT TO EXISTING LISTAND LF WILL POINT TO
C     THE NEW TOP OF LIST
      LF=LFREE
      XXY=XF(LFREE,1)
      LFREE=IFIX(XXY)
C
C**** CHECK IF LIST LNXT IS AN EMPTY LIST
C
      IF(LPT(LNXT).EQ.0)       GO TO 180
C
C**** LN IS THE POINTER WHICH POINTS TO PREVIOUS TOP OF
C     LIST
      LN=LPT(LNXT)
      LPT(LNXT)=LF
C
C**** LINK THE NEW ELEMENT WITH PREVIOUS TOP OF LIST
C
      XF(LF,1)=FLOAT(LN)
C
C**** PUT THE CURRENTLY GENERATED POINTS X(1),...,X(N1),
C     AND FUNCTION VALUES F(1),...,F(N2) IN XF
  150 DO 160 I=1,N1
      XF(LF,I+1)=X(I)
  160 CONTINUE
      DO 170 I=1,N2
      XF(LF,I+N1+1)=F(I)
  170 CONTINUE
```

```
      RETURN
C
C**** THE TOP OF LIST LNXT IS THE LF-TH ELEMENT IN THE FREE
      LIST
C
  180 LPT(LNXT)=LF
      XF(LF,1)=0.
      GO TO 150
C
C**** RELEASE LATEST LIST (RETURN DELETED)
C
  190 IF(LNXT.GT.L)          GO TO 210
  200 IF(LPT(LNXT).EQ.0)     GO TO 210
C
C**** LINK THE POINTERS OF THE LIST WHICH WE WANT TO
C     FREE WITH THE FREE LIST
      LF=LPT(LNXT)
      XXZ=XF(LF,1)
      LPT(LNXT)=IFIX(XXZ)
      LS=LFREE
      LFREE=LF
      XF(LF,1)=FLOAT(LS)
      GO TO 200
  210 LNXT=LNXT-1
      IFB=2
      IF(LNXT.GT.L)          GO TO 110
      LAST=LPT(LNXT)
  220 IF (LAST.EQ.0)         GO TO 110
C
C**** OBTAIN A POINT FROM THE LIST
C
      DO 225 I=1,N1
      XFF=XF(LAST,I+1)
      IF (A(I).GT.XFF.OR.B(I).LT.XFF) GO TO 226
  225 CONTINUE
```

```
      DO 230 I=1,N2
  230 F(I)=XF(LAST,N1+1+I)
      XYZ=XF(LAST,1)
      LAST=IFIX(XYZ)
      RETURN
  226 XYZ=XF(LAST,1)
      LAST=IFIX(XYZ)
      GO TO 220
      END
C
C**** COMPUTE NEW FUNCTION VALUES
C
      SUBROUTINE SYSTEM(A,B,N1,F,X,IFB)
      DIMENSION A(10),B(10),F(10),X(10),R(800)
      INTEGER NODES, IAM
      DATA NOLD/10317/
C
C**** USE RANDOM NUMBER GENERATOR TO GENERATE A
C     NUMBER BETWEEN 0 AND 1
      IAM=MYNODE()
      NODES=NUMNODES()
      IF(IFB.EQ.1)GO TO 170
      K=0
      DO 100 I=1,NODES
      DO 140 J=1,N1
      NEW=NOLD*65539
      IF(NEW)120,130,130
  120 NEW=NEW+2147483647+1
  130 R(K+1)=FLOAT(NEW)
      R(K+1)=R(K+1) *(0.4656613E-9)
      K=K+1
      NOLD=NEW
  140 CONTINUE
  100 CONTINUE
C
```

```
C**** OBTAIN RANDOM POINTS IN REGION
C
      DO 165 J=1,N1
      X(J)=R(IAM*N1+J)
      X(J)=(B(J)-A(J))*X(J)+A(J)
  165 CONTINUE
      GO TO 180
  170 DO 175 J=1,N1
      NEW=NOLD*65539
      IF(NEW)125,135,135
  125 NEW=NEW+2147483647+1
  135 S=FLOAT(NEW)
      S=S*(0.4656613E-9)
      NOLD=NEW
C
C**** OBTAIN RANDOM POINTS IN REGION
C
      X(J)=S
      X(J)=(B(J)-A(J))*X(J)+A(J)
  175 CONTINUE
C****USER MUST FURNISH EQUATIONS IN INCLDE FILE
  180 INCLUDE 'INPUT.INC'
      RETURN
      END
C
      SUBROUTINE-
      CSEND(MESG_TYPE,MESG_BUFF,SIZE,INODE,PID)
      INTEGER    MESG_TYPE,SIZE,PID,INODE,TID
      CHARACTER MESG_BUFF(*)
      PARAMETER(MNODES1=511)
      INTEGER TIDS
      COMMON/SK/ TIDS(0:MNODES1),NODES
      IF(INODE.EQ.-1) THEN
      FMCAST(NODES,TIDS(0) = MESG_TYPE,INFO)
      ELSE
```

```
      TID=TIDS(INODE)
      END IF
      RETURN
      END
C
      SUBROUTINE CRECV(MESG_TYPE,MESG_BUFF,SIZE,PID)
      INTEGER MESG_TYPE,SIZE,BUFID,PID
      CHARACTER MESG_BUFF(1)
      PARAMETER(MNODES1=511)
      INTEGER TIDS
      COMMON/SK/ TIDS(0:MNODES1),NODES
      PVMFRECV = (-1,MESG_TYPE,BUFID)
      FUNPACK = (STRING,MESG_BUFF,SIZE,1,INFO)
      RETURN
      END
C
      SUBROUTINE GDSUM(MESG_BUFF,N,TEMP)
      REAL*8 MESG_BUFF(1),TEMP(1)
      COMMON/PARALLEL/ NODES,IAM
      IF(NODES.EQ.1) RETURN
      IF(IAM.EQ.0) THEN
      DO IMESG=1,NODES-1
      CALL CRECV(1000,TEMP,8*N,0
      DO I=1,N
      MESG_BUFF(I)=MESG_BUFF(I)+TEMP(I)
      END DO
      END DO
      DO IMESG=1,NODES-1
      CALL CSEND(2000,MESG_BUFF,8*N,IMESG,0)
      END DO
      ELSE
      CALL CSEND(1000,MESG_BUFF,8*N,0,0)
      CALL CRECV(2000,MESG_BUFF,8*N,0)
      END IF
      RETURN
```

```
      SUBROUTINE GSYNC
     COMMON/PARALLEL/ NODES,IAM
      CHARACTER*1 DUMMY
     INODE=MOD(IAM+1,NODES)
     CALL CSEND(700,DUMMY,1,INODE,0)
     CALL CRECV(700,DUMMY,1,0)
      RETURN
     END
C
      SUBROUTINE GCOL(X,MSIZE,X_TOT)
     COMMON/PARALLEL/ NODES,IAM
      CHARACTER X(1),X_TOT(1)
     DO I=0,NODES-1
     II=I*MSIZE
     IF(I.EQ.IAM) THEN
     DO J=1,MSIZE
      X_TOT(II+J)=X(J)
     END DO
     ELSE
      CALL CSEND(500+IAM,X,MSIZE,I,0)
      CALL CRECV(500+I,X_TOT(II+1),MSIZE,0)
     END IF
     END DO
      RETURN
     END
C
      SUBROUTINE COLLECT(X,N,NPN,MSIZE)
     COMMON/PARALLEL/ NODES,IAM
      CHARACTER X(MSIZE,1)
      ISEND=IAM*NPN+1
     IRECV=1
      NSEND=NPN*MSIZE
      NRECV=NSEND
       IF(IAM.EQ.NODES-1) NSEND=MSIZE*(N-(NODES-1)*NPN)
     DO I=0,NODES-1
```

```
 IF(I.NE.IAM) THEN
 CALL CSEND(500+IAM,X(1,ISEND),NSEND,I,0)
  IF(I.EQ.NODES-1) NRECV=MSIZE*(N-(NODES-1)*NPN)
 CALL CRECV(500+I,X(1,IRECV),NRECV,0)
END IF
 IRECV=IRECV+NPN
END DO
 RETURN
END
```

APPENDIX B

PARALLEL IPSC/860

iPSC&2 and iPS08/860 User's Guide
Introduction

NOTE

In this manual, the terms "IPSC system" and "IPSC systems" refer to the iPSe/2, iPSC@12S, iPSCO/860, iPSCa/860S, and iPSe/86OPlus products.
The LPSC systems offer a solueion for Lirge-scale appLicaLions such as computational mechanics, petrolewn exploracion, electronic design, molecular modeling, and tactical and strategic systems. In an IPSC svstem, a large number of nodes work on the parts of a single problem.

IPSCOR SYSTEM HARDWARE

Figure I- 1 shows the two major hardware components of an IPSC system:

A front-end processor called the "host" (local system resource manager (SRvf) or remote workstation)

One or more cabinets (standard or compact) containing nodes (compute. I/O, and integrated) and storage devices (tape and disk drives)

The -Host-Provides Access to the Nodes

You communicate with the nodes in the cabinet through a front-end processor called the host. There are two kinds of hosts:
An IPSC system comes with a local host (the system resource manager, or SRM) that is connected directly to the IPSC nodes. You can also configure a workstation (by installing remote host software) to serve as a remote hoSL A remote host is connected to the SRM over an Ethernet link.

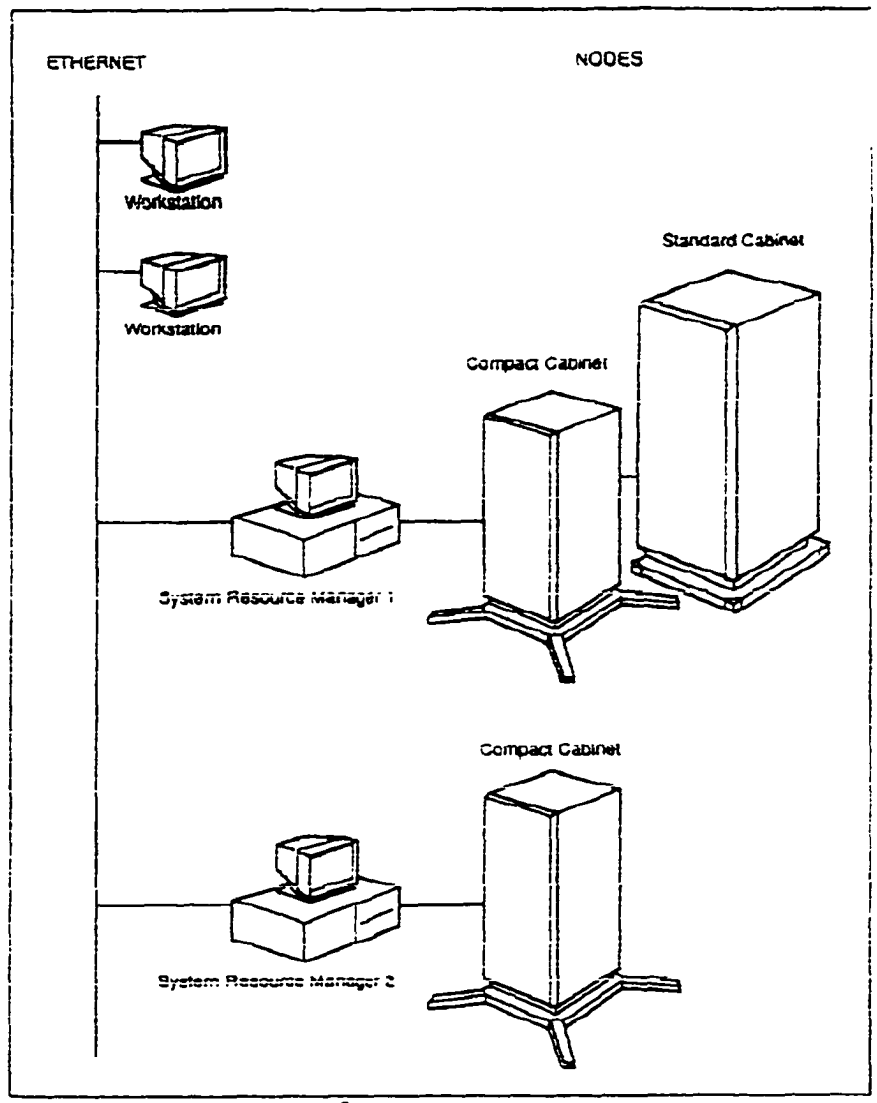


Figure 1-1. Two iPSC® Systems: One with One Compact Cabinet and One Standard Cabinet, and the Other with One Compact Cabinet

The host (local or remote) runs the UNIX operating system and provides the interface between you and the nodes. In the typical user model, you compile and link your host and node programs on the host, and then load and run your host programs on the host and your node programs on the nodes.

The -Cabinets-Contain Nodes and Storage Devices

The system cabinets contain cardcage modules (containing node boards) and peripheral modules (containing storage devices). There are two kinds of cabinets:

Standard Each standard cabinet can contain cardcage modules and peripheral modules configured in the following ways:

One cardcage module and four peripheral modules
Two cardcage modules and two peripheral modules
Four cardcage modules and no peripheral modules
The 17-slot cardcage holds the Unit Services Module (UMS) and up to 16 node boards,
VNIE Bus Interface Adapter (BIA) boards, or vector boards.

The peripheral module can hold up to four 760M-byte disk drives or 2G-byte tape drives.

Compact Each compact cabinet can contain only one 34-slot cardcage, which holds one USM and up to 32 compute nodes (one slot is reserved for future use).

NODES

A node is-a processor/memory pair.-Each node's memory is distinct from the host and from other nodes. Each node is full-Y connected to every other node throuh its Direct Connect Module (DCM). Because of

the DCIM, there is no "store and forward" of messages. Nodes that are nearest neighbors have nearly the same message latency as those that are not. Channel 7 has a special use. On all systems, channel 7 on node 0 connects to the SRM (giving the SRM the same kind of access to the DCM network that the nodes have). In addition, on a system that includes a Concurrent I/O (CIO) System, channel 7 on each I/O node is connected to a CIO Ethernet adapter.

There are four kinds of nodes:

Compute nodes

I/O nodes

Integrated nodes

Service nodes

Compute nodes and service nodes can reside in standard cabinets or in compact cabinets. I/O nodes and integrated nodes can reside only in standard cabinets.

Compute Nodes

Compute nodes are designed for performing computational tasks. There are five kinds of compute nodes:

CX nodes	These nodes are based on the Intel386TM processor and have an 80387 numeric coprocessor
SX nodes	These nodes are CX nodes in which the 80387 numeric coprocessor is replaced with an SX processor (for doing high-speed, floating-point, scalar arithmetic)
VX nodes	These nodes are CX nodes with a companion VX processor (for doing high-speed, vector arithmetic)
SXVX nodes	These nodes are SX nodes with a companion VX processor

RX nodes	These nodes are based on the i860TM microprocessor

NOTE

CX, SX, VX, and SXVX compute nodes are often collectively referred to as CX nodes because they are all based on the Intel386 microprocessor.

An iPSC/2 system contains no RX nodes; it consists entirely of CX, SX, VX, or SXVX compute nodes.

An iPSC/860 system either consists entirely of RX nodes or of a combination of all nodes (of which at least one is an RX node).

I/O Nodes

I/O nodes are based on CX compute nodes. Tle DCM of each I/O node connects to channel 7 of an associated compute or integrated node. There are two kinds of I/O nodes:

Standard I/O nodes are connected to disk and tape drives in the Concurrent I/O (CIO) system. These nodes have a SCSI (Small Computer System Interface) controller and a modified DCM (i.e., a DCM that has only channel 7).

Special I/O nodes are connected to the CIO Ethernet and VNM Bus Interface Adapter (BIA) options. These nodes are the same as a standard I/O node except that the SCSI controller is replaced with a PBX bus.

Integrated Nodes

Integrated nodes are Intel386-based nodes that combine compute and I/O node functions.

Service Nodes

Service nodes are (essentially) I/O nodes to which no disk is attached. The primary purpose of a service node is to run a node shell (**nsh**) without using any compute nodes. By default, when you invoke the nsh command, the node shell runs on a service node. The -s option to **nsh** makes the shell run on the current cube.

STORAGE DEVICES

An IPSC system can contain two kinds of storage devices:

> Disk drives for the Concurrent File System (CFS)
> Tape drives for system backups and installation of software

The Concurrent I/O (CIO) option provides each node with access to the Concurrent File System (CFS), whose array of disks appears as a single virtual disk that several node processes can access simultaneously. Available file storage is in excess of 4OG bytes.

The BIA board is an interface between an I/O (or integrated) node board with a PBX interface and a VUE controller board. (The VNE bus is a general purpose bus used to connect computer devices.) 'ne BIA resides in a slot adjacent to the node board, connected to the node board's PBX interface, and the VNE controller plugs into the BIA.

IPSC EXTENSIONS

fser-ver One or more file server processes are started automatically by getcube. A file server process allows the node to use standard I/O functions on the host. File server processes run on both the SRM and remote hosts.

lifeline The lifeline process monitors the existence of host user processes. When a host user process terminates, lifeline *notifies commser,* which then cleans up any appropriate data structures.*lifeline* runs on the SRM.

loader For CFS systems, the loader process runs on an 1/0 node and loads programs onto the cube.

loader.srm For non-CFS systems, the loader process runs on the SRM and loads programs onto the cube.

rcam The remote cube allocation manager process runs on the SRM and waits for a request from a remote host. When a remote host requests a cube, the two processes *tocube and tohost* are started. These processes handle message passing between the cube and the remote host. *rcam. tocube, and tohost run* on the SRM.

ripscd The remote IPSC daemon process handles remote requests (such as a remote compile) from the remote host. *ripscd* runs on the SRM.

adminproc For CFS systems, the administration process runs on an I/O node and keeps track of node processes.

tocube The tocube process handles message passing between the remote host and the SRM.

tohost *The tohost* process handles message passing between the SRM and the remote host.

TCP/IP NETWORKING SOFTWARE

The SRM and remote workstations also run the TCP/IP networldng software, which lets you remotely log into other systems on the network and transfer files between systems.

Optionally, the SRM and remote workstations can run the NFS networking software. NFS allows files to be shared transparently over a local network, making it possible to access files on other workstations as though they resided on your workstation (eliminating the need to log into another system and copy the files).

The CIO Ethernet option consists of TCP/IP running on the nodes of the system, and X Window client libraries that allow you to run and develop X Window applications.

IPSCO Software Development Environment

The IPSC software development environment includes:

Language processors
Remote host software
Interacbve parallel debugger

LANGUAGE PROCESSORS

The IPSC system supports a number of Ian uage processors that, depend g on the specific application, can be run on the SRM, a remote host, or another workstation. For example:

The Lisp user interface runs only on the SRM Host programs must be compiled and linked on the host on which they will ran.CX node programs can be compiled and linked on the SRM or on a remote host.

RX node programs can be compiled and linked on the SRM or a workstation (but not on a remote host).

REMOTE HOST SOFTWARE

Remote host software lets you use the facilities of the SRM without actually logging onto the SRM. For example, you can create a node source file on a remote host and then copied the executable file to SRM.

ALLOCATING AND RELEASING CUBES

IPSC system commands: attachcube
cubeinfo
getcu be
relcube

Before you can load programs onto the nodes, you must allocate a cube. The cube may consist of all the nodes in an IPSC system or a subset of Lhe nodes, but the number of nodes is always a power of two. That power is called the cube's dimension.

The examples in this section make a number of assumptions that may not be true about your particular IPSC system. For example, you can't try the examples that allocate vector processors unless vou have nodes equippe4d with that option.

If you decide to type in the examples as you read along, be sure to release cubes after allocating them. The nodes making up your cubes cannot be allocated by other users until you release them. To release a cube, use the relcube command. With the -a switch, relcube releases all your allocated cubes. For example,

relcube released xx cubes

where.xx represents the number of cubes that you have allocated. Most of the examples in this section end with a relcube just to ensure that, if you do type in the examples, you don't end up with unwanted cubes.

To allocate a cube, use the getcube command. If you use this command without arguments, you get the maximum number of available nodes. If your system contains both RX and CX nodes, getcube without arguments ignores the distinction and may allocate a mixed cube.

NOTE

If you allocate a cube containing a mixture of CX and RX nodes, you must load the correct type of executable programs on each type of node. That is, CX executables must be loaded on CX nodes and RX executables on RX nodes.

Redirecting Output

If you redirect the output of **getcube,** then the standard output or standard error of the node processes are also redirected. For example, the following command redirects the node's standard output to *myfile:*

getcube > *myfile*
relcube

relcube released 1 cube

Normally, getcube prints a response on the screen, indicating whether **getcube** was successful and how many nodes were allocated. In this example, however, the response is recorded in ***myfile***.

Number or Type of Nodes

Use getcube's -t switch to specify a particular number or type of nodes. For example, to allocate a 64-node CX cube, use the following command:

,getcube -t64(cx)

getcube successful: cube type 64m4cxnO allocated
relcube
relcube released 1 cube

The cube type 6 4m4 cx means 64 CX nodes, each with at least 4M bytes of memory was allocated. The n 0 indicates that the allocated cube consists of contiguous nodes beginning with the node physically located in slot 0.

getcube always allocates a cube whose dimension is a power of two. If you give something that is not a power of two, **getcube** rounds up. For example, the following command allocates a four-node cube, even though you specified three nodes:

Choosing the Node Memory Size

With getcube's -t switch, you can choose nodes with a particular amount of memory. For example, to allocate a four-dimensional cube (16 nodes) with at least 8M-byte nodes, use the following command:

getcube -td4m8

getcube successful: cube type d4m8cxnO allocated relcube
relcube released 1 cube

There's no guarantee that you will get only 8M-byte nodes. The specification is that you get at least 8M-byte nodes. That is, getcube may satisfy -td4cxm8 with 16M-byte nodes, even if 8M-byte nodes are available.

Choosing CX Nodes with VX Processors

A CX node that has a vector processor (called a VX node) may also have an SX processor and may have up to 8M bytes of memory.

To choose VX nodes, use getcube's -t switch. For example, the following command allocates a 16-node cube with VX nodes:

getcube -tl6vx

getcube successful: cube type 16m4vxnO allocated
relcube relcube released I cube
Because you didn't specify a memory size, you got whatever memory size is available. The 16m4vx in the getcube return message signifies that the minimum memory size of a node in your cube is 4M bytes, but you may have some nodes with more than 4M bytes.

Also note that if you didn't specify VX nodes, but VX nodes were all your system had, then getcube would give them to you. However if your system had a mixture of node types and you didn't specify VX nodes, your cube might end up with some VX nodes and some non-VX nodes - a type of hybrid cube.

RX nodes have their own vector capability and do not use a separate vector processor.

Choosing Hybrid Cubes

A hybrid cube is one whose nodes have different options or memory sizes. In some situations, you may want to specify a hybrid cube. Consider the following example. You want an eight-node cube, and you want at least four of those nodes to be VX nodes with 4M bytes. If you used *getcube -t8m4vx*, getcube would try for eight VX nodes (more than you need) and fail if they were unavailable. Instead, use the following command:

getcube -t4m4vx/4m4
getcube successful: cube type 4m4vxnO/4m4 allocated .relcube relcube released 1 cube

The / on the getcube command line separates the cube types. What if, when you used that last getcube, your IPSC system only had 8M-byte nodes? 8M bytes is certainly at least 4M bytes; so you'd still get a cube, but getcube's return message would be:

getcube successful: cube type 4m8vxnO/4m8 allocated

Specifying the Current Cube

The current cube is the last cube allocated. Without arguments, the cubeinfo command gives information about the current cube.

The relcube command also has a -c switch, allowing you to release a particular cube. If you have several allocated cubes and want to release all of them, use relcube's -a switch.

Here is an example that allocates two cubes and changes the current cube from the second to the first. Tle first cube takes the default name; the second is called *mycube*.

MANAGING PROCESSES

IPSC system commands:

killcube
load
startcube
waitcube

After you've allocated a cube, your next action is usually to load one or more pocesses onto the cube and run them. If you have RX nodes, you are restricted to one process per node, and its NX r)id must be 0.

To load a node process on the cube, use the **load** command. With no switches, **load** puts the **specified** file onto eveq node of the current cube and assigns an NX pid of 0 to the node process. Each node process starts running as soon as it's loaded. For example, if you have an executable called *node*, you can allocate a cube and load *node* on each node of that cube as follows:

getcube -t4

getcube successful: cube type 4m4cxnO allocated
load *node*
.relcube
relcube released 1 cube

Flushing and Canceling **Messages**

System Call	Environment
nushmsgo	host/node
msgcancelo	host/node

If after inspecting a pending message with the info ... 0 calls, you decide you don't want to receive it after all, you can get rid of it with flushmsgo. The **flushmsgo** call clears pending messages from the system buffer.

For example, here is a C code fragment that checks to see if a pending message has type 1; and if it does, flushes it. Otherwise, the program receives the message and continues.

```
long buf[1001, msg-type;


cprobe(-I);
msg_tvpe = infotypeo;
if (msg-type == 1)
                -Fl-ushmsg(l, mynodeo, mypido);
                                else
                crecv(msg-type, buf, sizeof(buf));
```

Note that flushmsgo only flushes messages pending to be received (that is, waiting in a system receive buffer), not those pending to be sent (that is, waiting in a system send buffer). That is, the call can be issued by either the sender or the receiver, but the parameters refer to the receiver. Specificaily, the node id in the second parameter specifies the destination node, not the source node; and the node pid in the third parameter specifies the destination pid, not the source pid.

'Me first parameter is really a @ selection parameter and not just a type. For example:

flushmsg(-l, mynodeo, mypido); *Cversion* call flushmsg(-l, mynodeo, mypido) *Fortran version*

would flush all messages waiting to be received by **mypido** on mynodeo.

msgcancelo
cancels an asynchronous send or receive operation. For example, assume that you post an asynchronous receive and then determine that you don't want the message. Issue a msgcancelo. When msgcancelo returns, you don't know whether the receive operation completed, but you do know that the application buffer that you set up for the received message is no longer in danger of being written into.

Table A-3. Summary of Routines for Cube Control (Fortran Version) (1 of 2)

Routine	Environment	Synopsis	Description
ATTACHCUBE()	Host	SUBROUTINE ATTACHCUBE(*cubename*) CHARACTER *cubename**(*)	Attach to a cube and make it the current cube.
CUBEINFO()	Host	INTEGER FUNCTION CUBEINFO(*ct*, *numslots*, *global*) CHARACTER*16 *ct*(*slotsize*, *numslots*) INTEGER *numslots*, *global*	Obtain information about allocated cubes.
FLICK()	Host, Node	SUBROUTINE FLICK()	On a node: Relinquish the CPU to another process. On the host: A no-op.
GETCUBE()	Host	SUBROUTINE GETCUBE(*cubename*, *cubetype*, *srmname*, *keep*) CHARACTER *cubename**(*) CHARACTER *cubetype**(*) CHARACTER *srmname**(*) INTEGER *keep*	Allocate a cube.
HANDLER()	Node	SUBROUTINE HANDLER(*etype*, *proc*) INTEGER *etype* EXTERNAL *proc* *proc* must be written in C. It has one argument: the hardware exception code.	Provide user-written exception handler for program failures.
KILLCUBE()	Host, Node	SUBROUTINE KILLCUBE(*node*, *pid*) INTEGER *node*, *pid*	Terminate and clear node process(es).
KILLPROC()	Host, Node	SUBROUTINE KILLPROC(*node*, *pid*) INTEGER *node*, *pid*	Terminate a node process.
KILLSYSLOG()	Host	SUBROUTINE KILLSYSLOG()	Terminate *syslog* process.
LOAD()	Host, Node	SUBROUTINE LOAD(*filename*, *node*, *pid*) CHARACTER *filename**(*) INTEGER *node*, *pid*	Load a node process.

```
prog:  pmfo.f
tk=.6
A=0
B=4
nodes=1
F(1)=X(1)-SIN(X(1)+X(2))+X(3)-1
F(2)=(X(2)-COS(X(1)-X(2)))**2+(X(3)-1)**2
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS   2.325094
              * NUMBER OF ROOTS ARE=        1
              * NUMBER OF NODES ARE=        1

                  ROOT BOUNDARIES FOUND

              X( 1) =   0.914063   TO   0.945313
              X( 2) =   1.000000   TO   1.007813
              X( 3) =   0.984375   TO   1.015625
```

```
prog:  pmfo.f
tk=.7
A=0
B=4
nodes=1
F(1)=X(1)-SIN(X(1)+X(2))+X(3)-1
F(2)=(X(2)-COS(X(1)-X(2)))**2+(X(3)-1)**2
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS   3.579084
              * NUMBER OF ROOTS ARE=        1
              * NUMBER OF NODES ARE=        1

                  ROOT BOUNDARIES FOUND

              X( 1) =   0.914063   TO   0.945313
              X( 2) =   1.000000   TO   1.007813
              X( 3) =   0.984375   TO   1.015625
```

```
prog:  pmfo.f
tk=.6
A=0
B=4
nodes=2
F(1)=X(1)-SIN(X(1)+X(2))+X(3)-1
F(2)=(X(2)-COS(X(1)-X(2)))**2+(X(3)-1)**2
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS  0.964323
              * NUMBER OF ROOTS ARE=        1
              * NUMBER OF NODES ARE=        2

                  ROOT BOUNDARIES FOUND

              X( 1) =    0.914063   TO    0.953125
              X( 2) =    1.000000   TO    1.007813
              X( 3) =    0.984375   TO    1.015625
```

```
prog:  pmfo.f
tk=.7
A=0
B=4
nodes=2
F(1)=X(1)-SIN(X(1)+X(2))+X(3)-1
F(2)=(X(2)-COS(X(1)-X(2)))**2+(X(3)-1)**2
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS   1.008329
              * NUMBER OF ROOTS ARE=        1
              * NUMBER OF NODES ARE=        2

                  ROOT BOUNDARIES FOUND

              X( 1) =    0.914063   TO    0.945313
              X( 2) =    0.984375   TO    1.007813
              X( 3) =    0.984375   TO    1.015625
```

```
prog:  pmfo.f
tk=.6
A=0
B=4
nodes=4
F(1)=X(1)-SIN(X(1)+X(2))+X(3)-1
F(2)=(X(2)-COS(X(1)-X(2)))**2+(X(3)-1)**2
          ***********************************************
       * CPU TIME FOR THIS SET OF FUNCTIONS IS  0.726655
                * NUMBER OF ROOTS ARE=       1
                * NUMBER OF NODES ARE=       2

                   ROOT BOUNDARIES FOUND

                X( 1) =    0.914063   TO    0.945313
                X( 2) =    1.000000   TO    1.007813
                X( 3) =    0.984375   TO    1.015625
```

```
prog:  pmfo.f
tk=.7
A=0
B=4
nodes=4
F(1)=X(1)-SIN(X(1)+X(2))+X(3)-1
F(2)=(X(2)-COS(X(1)-X(2)))**2+(X(3)-1)**2
          ***********************************************
       * CPU TIME FOR THIS SET OF FUNCTIONS IS   1.078191
                * NUMBER OF ROOTS ARE=       1
                 * NUMBER OF NODES ARE=  4

                   ROOT BOUNDARIES FOUND

                X( 1) =    0.914063   TO    0.945313
                X( 2) =    1.000000   TO    1.007813
                X( 3) =    0.984375   TO    1.015625
```

```
prog: pmfo.f
tk=.6
A=-10
B=10
nodes=1
F(1)=X1-X2**2+2
F(2)=-X1**2+X2+X3
F(3)=2*X1-X2**2+X3
```

****

* CPU TIME FOR THIS SET OF FUNCTIONS IS 8.072931
* NUMBER OF ROOTS ARE= 3
* NUMBER OF NODES ARE= 1

ROOT BOUNDARIES FOUND

X(1) =	1.530762	TO	1.538086
X(2) =	1.877441	TO	1.884766
X(3) =	0.458984	TO	0.488281

ROOT BOUNDARIES FOUND

X(1) =	0.341797	TO	0.356445
X(2) =	-1.538086	TO	-1.528320
X(3) =	1.645508	TO	1.660156

ROOT BOUNDARIES FOUND

X(1) =	-2.011719	TO	-1.992188
X(2) =	-0.019531	TO	0.009766
X(3) =	3.984375	TO	4.042969

```
prog:  pmfo.f
tk=.6
A=-10
B=10
nodes=2
F(1)=X1-X2**2+2
F(2)=-X1**2+X2+X3
F(3)=2*X1-X2**2+X3
```

****

* CPU TIME FOR THIS SET OF FUNCTIONS IS 5.399231
* NUMBER OF ROOTS ARE= 3
* NUMBER OF NODES ARE= 2

ROOT BOUNDARIES FOUND

X(1) =	0.336914	TO	0.356445
X(2) =	-1.538086	TO	-1.523438
X(3) =	1.640625	TO	1.665039

ROOT BOUNDARIES FOUND

X(1) =	-2.001953	TO	-1.992188
X(2) =	-0.019531	TO	0.009766
X(3) =	3.984375	TO	4.023438

ROOT BOUNDARIES FOUND

X(1) =	-1.887207	TO	-1.875000
X(2) =	-0.351563	TO	-0.332031
X(3) =	3.867188	TO	3.891602

```
prog:  pmfo.f
tk=.6
A=-10
B=10
nodes=4
F(1)=X1-X2**2+2
F(2)=-X1**2+X2+X3
F(3)=2*X1-X2**2+X3
          ************************************************
        * CPU TIME FOR THIS SET OF FUNCTIONS IS   3.978216
                * NUMBER OF ROOTS ARE=        3
                * NUMBER OF NODES ARE=        4

                    ROOT BOUNDARIES FOUND

                X( 1) =    0.336914   TO    0.356445
                X( 2) =   -1.538086   TO   -1.523438
                X( 3) =    1.640625   TO    1.665039

                    ROOT BOUNDARIES FOUND

                X( 1) =   -2.001953   TO   -1.992188
                X( 2) =   -0.019531   TO    0.009766
                X( 3) =    3.984375   TO    4.023438

                    ROOT BOUNDARIES FOUND

                X( 1) =   -1.887207   TO   -1.875000
                X( 2) =   -0.351563   TO   -0.332031
                X( 3) =    3.867188   TO    3.891602
```

```
prog: pmfo.f
tk=.6
A=-5
B=10
nodes=1
F(1)=2*X1+2*X2+3*X3+2*X4+2
F(2)=4X1-3*X2+X4+7
F(3)=6X1+X2-6X3-5*X4-6
F(4)=-8*X1+X2*EXP**X3+X4**2+4*X4-EXP**X3
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS   34.85888
              * NUMBER OF ROOTS ARE=        2
              * NUMBER OF NODES ARE=        1

                   ROOT BOUNDARIES FOUND

              X( 1) =   -0.473633   TO   -0.451660
              X( 2) =    0.705566   TO    0.771484
              X( 3) =    1.020508   TO    1.203613=
              X( 4) =   -3.051758   TO   -2.832031

                   ROOT BOUNDARIES FOUND

              X( 1) =   -0.502930   TO   -0.497437
              X( 2) =    0.976563   TO    1.005859
              X( 3) =    0.321045   TO    0.390625
              X( 4) =   -2.070313   TO   -1.982422
prog: pmfo.f
tk=.6
A=-5
B=10
nodes=2
F(1)=2*X1+2*X2+3*X3+2*X4+2
F(2)=4X1-3*X2+X4+7
F(3)=6X1+X2-6X3-5*X4-6
F(4)=-8*X1+X2*EXP**X3+X4**2+4*X4-EXP**X3
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS   11.34062
              * NUMBER OF ROOTS ARE=        2
              * NUMBER OF NODES ARE=        2

                   ROOT BOUNDARIES FOUND
```

X(1) = -0.473633 TO -0.451660
X(2) = 0.705566 TO 0.771484
X(3) = 1.020508 TO 1.203613
X(4) = -3.051758 TO -2.832031

ROOT BOUNDARIES FOUND

X(1) = -0.502930 TO -0.497437
X(2) = 0.976563 TO 1.005859
X(3) = 0.321045 TO 0.390625
X(4) = -2.070313 TO -1.982422

prog: pmfo.f
tk=.6
A=-5
B=10
nodes=4
F(1)=2*X1+2*X2+3*X3+2*X4+2
F(2)=4X1-3*X2+X4+7
F(3)=6X1+X2-6X3-5*X4-6
F(4)=-8*X1+X2*EXP**X3+X4**2+4*X4-EXP**X3

****

* CPU TIME FOR THIS SET OF FUNCTIONS IS 10.28291
* NUMBER OF ROOTS ARE= 2
* NUMBER OF NODES ARE= 4

ROOT BOUNDARIES FOUND

X(1) = -0.473633 TO -0.451660
X(2) = 0.705566 TO 0.771484
X(3) = 1.020508 TO 1.203613
X(4) = -3.051758 TO -2.832031

ROOT BOUNDARIES FOUND

X(1) = -0.502930 TO -0.497437
X(2) = 0.976563 TO 1.005859
X(3) = 0.321045 TO 0.390625
X(4) = -2.070313 TO -1.982422

```
prog:  pmfo.f
tk=.6
A=-3
B=6
nodes=1

F(1)=2*X(1)-X(2)+4*X(3)-3*X(4)+X(5)-11
F(2)=-X(1)+X(2)+2*X(3)+X(4)+3*X(5)-7*X(6)
F(3)=4*X(1)+2*X(2)+3*X(3)+3*X(4)-X(5)-X(6)**2
F(4)=-3*X(1)+X(2)+3*X(3)+2*X(4)+4*X(5)-2*X(6)**3
F(5)=X(1)+3*X(2)-X(3)+4*X(4)+4*X(5)-9*X(6)
F(6)=-4*X(1)+2*(X(2)*X(4))+X(3)*(2**X(2))+X(5)**2-2*X(5)
```

****

* CPU TIME FOR THIS SET OF FUNCTIONS IS 275.2281

* NUMBER OF ROOTS ARE= 2

* NUMBER OF NODES ARE= 1

ROOT BOUNDARIES FOUND

X(1) =	0.867188	TO	0.928711
X(2) =	3.038086	TO	3.187500
X(3) =	0.875977	TO	0.919922
X(4) =	-1.699219	TO	-1.646484
X(5) =	3.679688	TO	3.750000
X(6) =	1.921875	TO	1.957031

ROOT BOUNDARIES FOUND

X(1) =	1.429688	TO	1.500000
X(2) =	-1.558594	TO	-1.453125
X(3) =	1.113281	TO	1.218750
X(4) =	0.937500	TO	1.025391
X(5) =	4.769531	TO	4.875000
X(6) =	2.080078	TO	2.132813

```
prog:  pmfo.f
tk=.6
A=-3
B=6
nodes=2

F(1)=2*X(1)-X(2)+4*X(3)-3*X(4)+X(5)-11
F(2)=-X(1)+X(2)+2*X(3)+X(4)+3*X(5)-7*X(6)
F(3)=4*X(1)+2*X(2)+3*X(3)+3*X(4)-X(5)-X(6)**2
F(4)=-3*X(1)+X(2)+3*X(3)+2*X(4)+4*X(5)-2*X(6)**3
F(5)=X(1)+3*X(2)-X(3)+4*X(4)+4*X(5)-9*X(6)
F(6)=-4*X(1)+2*(X(2)*X(4))+X(3)*(2**X(2))+X(5)**2-2*X(5)
```

****

* CPU TIME FOR THIS SET OF FUNCTIONS IS 100.1219

* NUMBER OF ROOTS ARE= 2

* NUMBER OF NODES ARE= 2

ROOT BOUNDARIES FOUND

X(1) =	0.867188	TO	0.928711
X(2) =	3.038086	TO	3.187500
X(3) =	0.875977	TO	0.919922
X(4) =	-1.699219	TO	-1.646484
X(5) =	3.679688	TO	3.750000
X(6) =	1.921875	TO	1.957031

ROOT BOUNDARIES FOUND

X(1) =	1.429688	TO	1.500000
X(2) =	-1.558594	TO	-1.453125
X(3) =	1.113281	TO	1.218750
X(4) =	0.937500	TO	1.025391
X(5) =	4.769531	TO	4.875000
X(6) =	2.080078	TO	2.132813

```
prog:  pmfo.f
tk=.6
A=-3
B=6
nodes=4

F(1)=2*X(1)-X(2)+4*X(3)-3*X(4)+X(5)-11
F(2)=-X(1)+X(2)+2*X(3)+X(4)+3*X(5)-7*X(6)
F(3)=4*X(1)+2*X(2)+3*X(3)+3*X(4)-X(5)-X(6)**2
F(4)=-3*X(1)+X(2)+3*X(3)+2*X(4)+4*X(5)-2*X(6)**3
F(5)=X(1)+3*X(2)-X(3)+4*X(4)+4*X(5)-9*X(6)
F(6)=-4*X(1)+2*(X(2)*X(4))+X(3)*(2**X(2))+X(5)**2-2*X(5)
          ***********************************************
      * CPU TIME FOR THIS SET OF FUNCTIONS IS   82.53902
                * NUMBER OF ROOTS ARE=        2
                * NUMBER OF NODES ARE=        4

                ROOT BOUNDARIES FOUND

              X( 1) =    0.867188   TO    0.928711
              X( 2) =    3.038086   TO    3.187500
              X( 3) =    0.875977   TO    0.919922
              X( 4) =   -1.699219   TO   -1.646484
              X( 5) =    3.679688   TO    3.750000
              X( 6) =    1.921875   TO    1.957031

                ROOT BOUNDARIES FOUND

              X( 1) =    1.429688   TO    1.500000
              X( 2) =   -1.558594   TO   -1.453125
              X( 3) =    1.113281   TO    1.218750
              X( 4) =    0.937500   TO    1.025391
              X( 5) =    4.769531   TO    4.875000
              X( 6) =    2.080078   TO    2.132813
```

APPENDIX C

IMPLEMENTATION

CERENKOV COUNTERS

A method of producing visible radiation was discovered by P. A. Cerenkov . He observed that a beam of fast electrons, such as beta particles from radioactive substances, when moving in a transparent medium caused the emission of visible radiation, provided that the velocity of the electrons was greater than the velocity of light in the same medium. The theory developed by I. M. Frank and I. E. Tamm predicts that the light should be propagated at an angle zero to the direction of motion of the electron given by.

$$\cos\theta = \frac{c}{nv}$$

An electron moving through a substance loses most of its energy in ionization and excitation of the atoms. In these processes the electron itself experiences small accelerations, hence radiates energy in the form of electro.

The direction of motion of the electrons is in 5 transparent medium magnetic waves. Since these waves originate at different points along the path of the electron, radiant energy will be observed only if the waves from the different points reinforce each other.

The condition for reinforcement of the waves by interference can readily be derived with the aid of Figure 4.2. The electron beam enters the transparent medium at M and continues along the path MAB. The electron radiates energy in all directions from the points along this path.

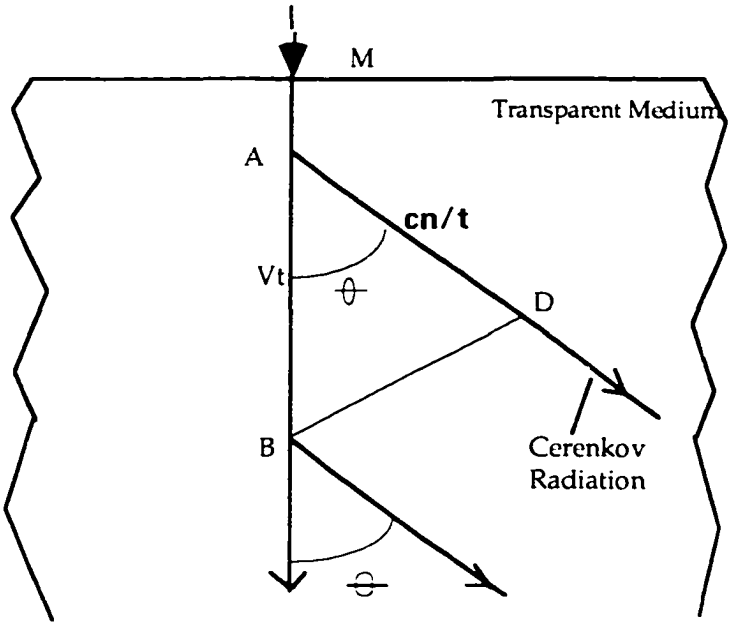


Figure C.1 Direction of propagation of Cerenkov radiation relative to

From the figure above it follows that the radiation will travel in a direction such that.

$$\cos\theta = \frac{c}{nv}\cos$$

also the intensity of Cerenkov radiation emitted is given by the expression

$$I(v)dv = \frac{4p^2ke^2Z^2}{c^2}(1-\frac{c^2}{v^2n^2})vdv$$

where Ze is the electric charge of the particle in coulombs, I dv is the number of joules emitted per meter of path for frequencies between v and v + dv.

Cerenkov radiation is useful in the detection of highenergy charged partides. The only condition imposed on these particles is that their speeds must be greater than the speed of light in the same medium. In any given medium the angle of emission 0 is determined only by the velocity of the particle; hence the detection of the Cerenkov radation is a convenient method for measuring d the velocities of highenergy charged particles. As Equation above shows, the intensity of the radiation is proportional to the square of the electric charge, so that an absolute charge measurement is possible.

In practice many substances have been used as Cerenkov detectors. In conjunction with a number of photomultiplier tubes, the entire rrangement is called a Cerenkov counter. Lead glass and polymers such as lucite have been used as the medium, as have liquids such as water, and CCl_4. In each case convenience and the index of refraction are the important considerations. Under such circumstances various gases are suitable for Cerenkov counters; N_2, CO_2, and SFs are often used; for instance, it is about 1.1 for CO_2 at 200 atmospheres of pressure. By varying the pressure, n can be adjusted within a continuous range of values, leading to a highly desirable flexibility of the instrument.

SUPER-KAMIOKANDE

Super-Kamiokande is a 50,000 ton water Cherenkov detector. This large volume water detectors were invented to discover proton decay. As Super-K is 6-10 times larger than the previous generation of detectors, it can reach a proton lifetime of 10^{34} years, reaching

predictions by supersymmetric models (which are themselves supported by the apparent convergence of running coupling constants as measured at LEP and elsewhere).

The Super-K is designed to investigate contained events of energy 1 GeV, and also it will naturally pursue the intriguing puzzle of atmospheric neutrinos first observed in the prior generation of water Cherenkov detectors. These neutrinos are produced in the shower of particles caused by cosmic ray interactions in our atmosphere. The puzzle is that the ratio of the flux of electron neutrinos to muon neutrinos disagrees with our best understanding of the underlying processes. The most interesting explanation may be that the neutrinos have a small but finite mass and undergo flavor oscillation as they travel the long distance to the detector.

Super-Kamiokande has been instrumented with these low energy interactions in mind, and will record 5,000 solar neutrino interactions per year. And perhaps neutrinos from another- dying- sun will be detected.

PHOTOMULTIPLIER

A photon enters the photomultiplier through the glass surface and hits the photo-cathode, which is placed on the inner surface of the glass (see appendex 4). The photo-cathode hit by the photon emits an electron. A photo-cathode is just a special substance which is very efficient at doing this. Inside the photomultiplier there is a vacuum.

The electron is attracted and accelerated to the first dynode, which is charged positively by a high voltage. As it hits it with great energy, the dynode emits several electrons, which are then attracted to

a second dynode, which has even higher positive electric potential. This process repeats many times. At the last dynode we have a really huge number of electrons. This way the signal of a single electron was enormously amplified. This is why photomultipliers are such sensitive light detectors.

ELECTRONICS

Modern detectors provide a variety of information on the detected particles in the form of electrical signals. In order to extract this information, the signal must be further processed by an computer and electronic systems. This system can be designed to perform a enormous variety of tasks.

For example sorting out the various signals, extracting energy, timing, position between two signals, and based on this information, make a decision as to the acceptability of event and take appropriate action. This system should virtually run it self by activating a recording or other devices.

The first standard established for high energy physics is a modular system called NIM (Nuclear Instrumentation Module). in this system, the basic electronic apparatus, like discriminator, logic unit, etc., are constructed in the form of modules according to standard mechanical and electrical specifications. In turn, fit into standardized bins which supply the modules with standard power voltages.

Any NIM module will fit into any NIM bin. The standard NIM bin is constructed to accept up to 12 single width modules the external bin dimensions are such as to allow mounting in 19 inch racks or cabinets. the rear power connectors it provide at the very least six

standard dc voltages, which are -6 V, +6 V -12 V, +12 V, -24 V, +24 V. NIM modules include both analog and digital instruments. First the slow positive signal and fast negative signal (NIM logic) also include TTL (transistor-transistor logic) and ECL (emitter coupled logic) table B-1 show the input and output for logic 0 , and 1.

Table B-1 NIM logic input and output

TYPE	ON/OFF	OUTPUT	INPUT
S.P.	LOGIC 1	+4 TO +12 V	+3 TO +12 V
	LOGIC 0	+1 TO -2 V	+1.5 TO -2 V
F.N.	LOGIC 1	-14 TO -18 mA	-12 TO -36 mA
	LOGIC 0	-1 TO +1 mA	-4 TO +20 mA
TTL	LOGIC 1	2 TO 5 V	SAME
	LOGIC0	0 TO 0.8 V	SAME
ECL	LOGIC 1	-1.75 V	----
	LOGIC 0	- 0.90 V	----

TIME TO DIGITAL CONVERTER (TDC)

This CAMAC module is used to obtain a time measurement in digital form between two signals. The basic principle here is to use the start signal to gate on a scalar which counts at a constant frequency oscillator (clock). At the arrival of a second stop signal, this scalar is gate

off to yield a number proportional to the time interval between the pulses. Figure C-2 shows block diagram for this module.

The time are measured with respect to a common reference time mark which can occur before or after the individual time signal input to be measured. This common reference concept is appropriate when the relative times between one input channel and any other input channel is of interest, since they are both measured with respect to a common time.

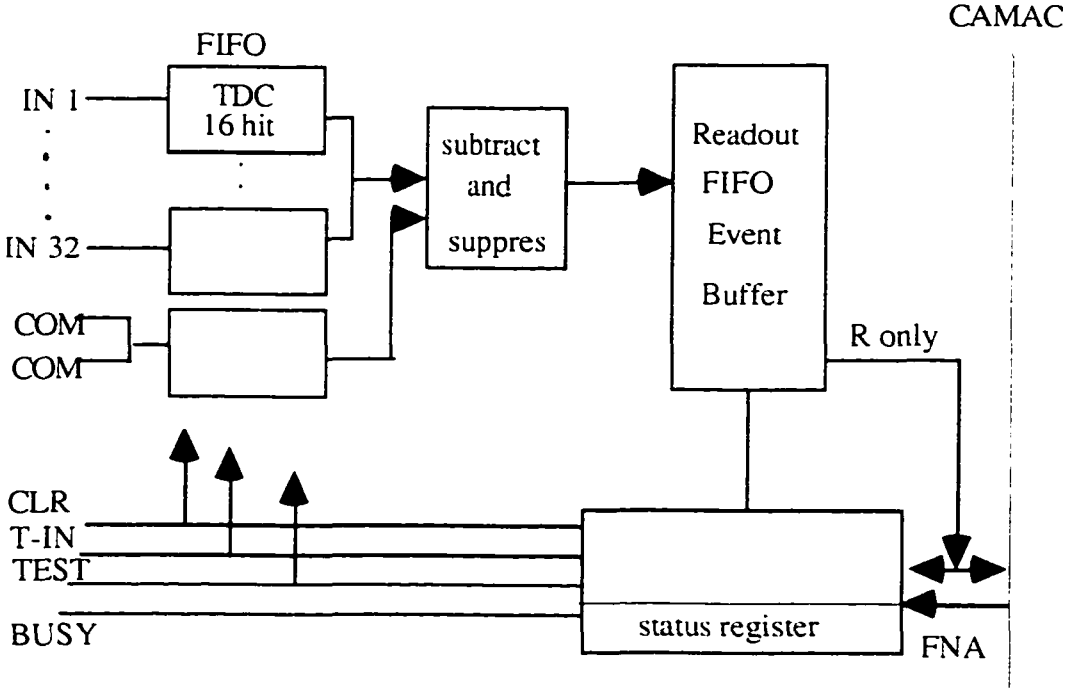


Figure C-2 TDC block diagram

This module must be configured and operated from CAMAC. The configuration of the TDC is determined by the status register and data value stored in the register is decoded as a bit pattern. The bit pattern permits the mode selection, internal time window duration, edge sensitivity, test features and trigger output type.

DISCRIMINATORS

The discriminator is a device which responds only to input signals with a pulse height greater then a certain threshold value. If this criterion is satisfied, as shown in figure C-3. the discriminator responds by issuing a standard logic signal.

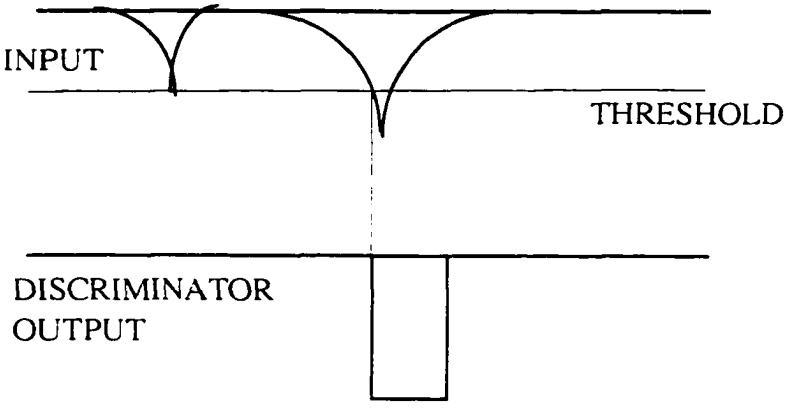


Figure C-3 Discriminator operation.

Normally this module is composed of eight channels with three basic sections as shown in figure C-4. input and output and timing. The input stage is based on the LeCroy model LD601 hybrid. this unit contains all of the circuitry of discriminator with the exception of the

input termination and high voltage protection. The output of the LD601 drives a fast NPN switching transistor. Normally, this transistor is found in its cut off region and the collector is clamped to approximately 400 mV by a Schottky diode.

The output stage is a simple NPN inverter stage with the emitter returned to the negative 3.8 volt supply. The stage is normally biased in the cut off region. During an output pulse, there is sufficient base current supplied from the LD601 to turn the output driver on. In the on condition, the collector voltage is approximately about the -2.7 volt.

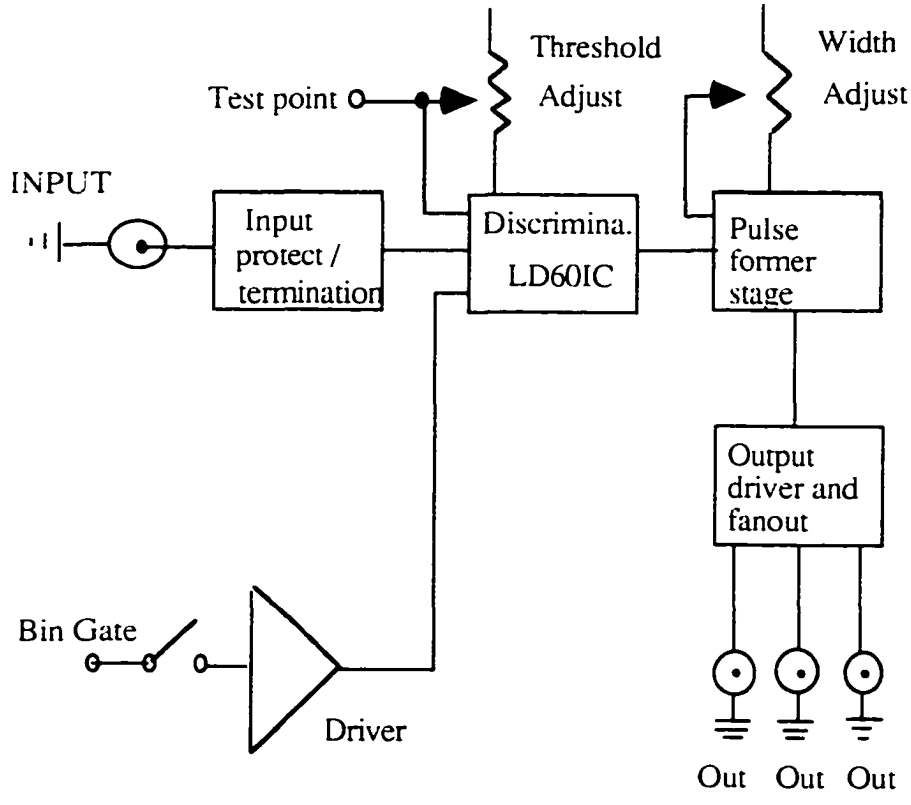


Figure C-4 One of eight discriminator channels

MAJORITY LOGIC UNIT.

The coincidence unit determines if two or more logic signals are coincidence in time and generates a logic signal if true and no signal if false as shown in figure C-5. This unit is one example of a more general class of units known as the logic gate. These are units which perform the equivalent of Boolean logic operation on the input signals. One should look at NIm for more information.

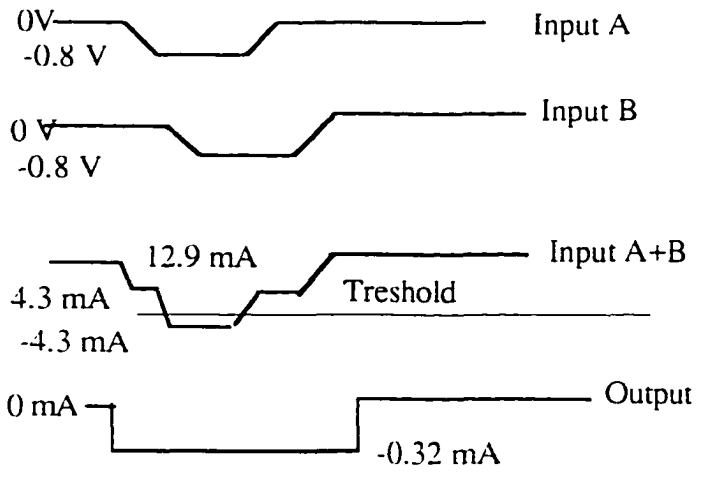


Figure C-5 The summing method for two input

This unit provides the function of fan in, coincidence, inhibit, and majority logic with high fan out capability along with 150 MHZ operation. Each of two identical channels accepts standard NIM logic signals and all inputs are terminated in 50 ohm connectors.

LOGIC FAN/IN FAN/OUT

Fan outs are active circuits which allow the distribution of one signal to several parts of an electronics system by dividing the input signal into several identical signals of the same height and shape as shown in figure C-6. The fan in on the other hand accept several input signal and delivers the algebraic sum at the output.

The quad fan in fan out is a multi-functional fast logic module with four inputs which accept both NIM and TTL levels. In this module there are four FET switches to couple the channels into two groups of two or one group of four. When cross connected, the output emitter followers of the input stage are emitter ORed, so any one can drive the common (cross bus).

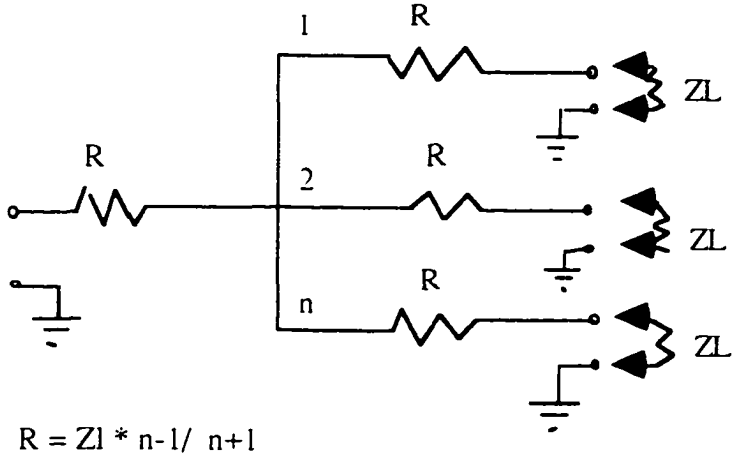


Figure C-6 Fan out circuit.

SKMAN.FOR

This driver (program) is used to read STXT data events from YMCA Monte-Carlo output files and event data from detector. Then it feed them to root isolation program.

```
      program skman
       implicit none
      integer count,nbig
      integer istatus
       real pmt_position(3,14000)
       integer pmt_type(14000),n_pmts
      real pmt_dat(7,1000)
        integer n_hits,event_number,nhits0
        common /pmt_data/ pmt_position,pmt_type,n_pmt
       common/hay1/ n_hits,pmt_dat
       common/yhay1/event_number
c
        open(unit=10,file='out.dat',status='new',recl=1024,
      $access='sequential',form='formatted')
      count = 0
      call read_coord
       call read_input_file
 1     call read_stxt_event(istatus)
      if(istatus.eq.-1)then
       goto 5
      else if (istatus.eq.2)then
         print*,' ERROR:  Was not able to read event: ',event_number
        goto 1
      else if(istatus.eq.0)then
        if (istatus.eq.1)then
           print*,' WARNING:  May be a SUSPECT event: ',
event_number
        end if
        count = count+1
        if(n_hits.gt.500)then
```

```
        nbig=nbig+1
        goto 1
       end if
        nhits0=n_hits  !  nhits before Hayai cleaning
       goto 1
     else
       print*,'  ERROR:  Non-Standard istatus: ',event_number
     end if
 5   continue
     end
c
     subroutine read_coord
     character*72 coor_file,comment
     integer i,j,type,pmt_number
     real position(3)
     real pmt_position(3,14000)
      integer pmt_type(14000),n_pmts,n_pmts_max
      common /pmt_data/ pmt_position,pmt_type,n_pmts
     n_pmts_max=14000
      open(unit=8,name='sk_coor_file.dat',status='old',err=13)
     read(8,'(a)') comment
     n_pmts=0
     do i=1,n_pmts_max
       read(8,1000,err=14,end=15) pmt_number,type,position
1000    format(1x,i5,1x,i1,1x,3(f10.2,1x))
      pmt_type(i)=0
      do j=1,3
           pmt_position(j,i)=0.
            if((pmt_number.ge.1).and.(pmt_number.le.n_pmts_max))
       then
            pmt_position(j,pmt_number)=position(j)
           pmt_type(pmt_number)=type
           n_pmts=n_pmts+1
          else
```

```
            print *,'Error in PMT coordinate file at ',pmt_number
          stop
           end if
        end do
      end do
 15   continue              !normal read
      n_pmts=n_pmts/3
     close(8)
     return
 13   continue              !error in opening file
      print *,'Error in Opening Coordinate File'
      print *,coor_file
     goto 14
 14   continue
      print *,'Error in reading coordinate file '
     stop
      end
c
c     Subroutine to find file name and open it.
c
      subroutine read_input_file
     character ifile*72
      open(unit=8,name='evnt.stxt',status='old',err=3)
     ifile_active=.true.
     return
 2    continue
      print *,'Error in reading input file name'
 3    continue
      print *,'Error in opening input file'
     end
c
c     routine to read a McGrew style ASCII event.
c
     subroutine read_stxt_event(istatus)
      implicit none
```

```
 common /pmt_data/ pmt_position,pmt_type,n_pmts
 common/hay1/ n_hits,pmt_dat
 real pmt_position(3,14000)
 integer pmt_type(14000),n_pmts
 real pmt_dat(7,1000)
 integer n_hits,event_number
 integer number_part,part_id(100)
 real part_pos(3,100),part_dir(3,100),part_mom(100),part_time(100)
 integer N_PMTS_MAX
 integer tube_number
 real time,pulse_height
 integer istatus          !0=got an event
 character line*80,blank_line*80
 logical evt_active       !started an event
 logical tdata_active     !now reading tube data
 logical lost             !true if we are lost
 logical first            !first time through this routine
 logical lsu_simulation
 data first/.true./
 data evt_active/.false./
 data tdata_active/.false./
 data lost/.false./
 integer i
 integer ptr_first        !pointer to first non-blank character
 integer ptr1             !pointer to item in line being read
 integer ptr2             !pointer to item in line being read
 integer n_lines          !# lines with current identifier
 character major_id*3,minor_id*3
 real charge,ydir(3),yver(3)
 N_PMTS_MAX=14000
 if(first) then
  do i=1,80
    blank_line(i:i)=' '
  end do
  first=.false.
```

```
    end if
1   continue                !expect major ID at this point
    line=blank_line
    read(8,1000,end=99,err=13) line
1000 format(a)
    do i=1,80               !look for first non-blank character
      if(line(i:i).ne.' ') goto 2
    end do
    goto 1                  !a blank line, ignore it
2   continue
    ptr_first=i             !set up ptr1 on forst non-blank character
    ptr1=ptr_first+2        !set up ptr 2 2 characters over
     if(line(ptr_first:ptr1).eq.'+++') then !special line identifier
      lost=.false.          !we are not lost
      ptr1=ptr1+1           !now look for major identifier
        do while(line(ptr1:ptr1).eq.' ') !move to next non-blank
       ptr1=ptr1+1
       if(ptr1.ge.80) then
          print *,'Blank record ID line'
         print *,line
         goto 1             !got a blank record id line somehow
       end if
      end do
      ptr2=ptr1+2           !set up ptr2 two characters over
       major_id=line(ptr1:ptr2) !read major ID from line
      ptr1=ptr2+1               !now move over to minor ID
      ptr2=ptr1+2
       minor_id=line(ptr1:ptr2) !next three better be minor id
      ptr1=ptr2+1               !set up on next character
       if(major_id.eq.'EVT') then
         if(minor_id.eq.'BEG') then
          if(evt_active) then
             print *,'warning: EVTBEG without EVTEND'
         end if
          evt_active=.true. !set active and read number of lines
```

```
          read(line(ptr1:80),*,err=113,end=113) n_lines
           n_lines=n_lines-1 !number of lines still to read
           if(n_lines.ge.1) then !now read event number
            read(8,*,end=113,err=113) event_number
            n_lines=n_lines-1
          end if
           do while(n_lines.ge.1)
            read(8,1000,end=113,err=113) line !skip over rest
             n_lines=n_lines-1
          end do
          goto 1          !ready for next major id
113         continue        !come here for error reading
           print *,'Error in EVTBEG lines'
          print *,line
          goto 1
           elseif(minor_id.eq.'END') then
           if(.not.evt_active) then
             print *,'warning: EVTEND without EVTBEG'
          end if
           evt_active=.false.
           read(line(ptr1:80),*,err=114,end=114) n_lines
           n_lines=n_lines-1
           do while(n_lines.ge.1)
            read(8,1000,end=114,err=114) line !skip over rest
             n_lines=n_lines-1
          end do
          istatus=0    !OK
          return          !got an event
114         continue
           print *,'Error in EVTEND lines'
          print *,line
          istatus=1    !warning, though we think we got an event
          return
        else
           print *,'Unrecognized EVT minor ID ',minor_id
```

```
        read(line(ptr1:80),*,err=1,end=1) n_lines
        do while(n_lines.ge.1)
          read(8,end=1,err=1) line !skip over rest
           n_lines=n_lines-1
        end do
        goto 1
      end if
      elseif(major_id.eq.'DET') then
        read(line(ptr1:80),*,err=213,end=213) n_lines
        n_lines=n_lines-1
        do while(n_lines.ge.1)
          read(8,end=213,err=213) line !skip over rest
           n_lines=n_lines-1
      end do
      goto 1
213    continue
        print *,'Error in DET line'
       print *,line
      goto 1
      elseif(major_id.eq.'CAL') then
        read(line(ptr1:80),*,err=313,end=313) n_lines
        n_lines=n_lines-1
        do while(n_lines.ge.1)
          read(8,end=313,err=313) line !skip over rest
           n_lines=n_lines-1
      end do
      goto 1
313     continue
        print *,'Error in CAL line'
       print *,line
      goto 1
      elseif(major_id.eq.'CLK') then
        read(line(ptr1:80),*,err=413,end=413) n_lines
        n_lines=n_lines-1
        do while(n_lines.ge.1)
```

```
          read(8,end=413,err=413) line !skip over rest
           n_lines=n_lines-1
        end do
        goto 1
413       continue
         print *,'Error in CLK line'
         print *,line
        goto 1
        elseif(major_id.eq.'COM') then
         read(line(ptr1:80),*,err=513,end=513) n_lines
          n_lines=n_lines-1
         do while(n_lines.ge.1)
           read(8,end=513,err=513) line !skip over rest
            n_lines=n_lines-1
        end do
        goto 1
513       continue
         print *,'Error in COM line'
         print *,line
        goto 1
        elseif(major_id.eq.'KIN') then
         read(line(ptr1:80),*,err=613,end=613) n_lines
          n_lines=n_lines-1
         number_part=0
         do while(n_lines.ge.1)
            if(lsu_simulation) then
              number_part=number_part+1
              read(8,*,end=613,err=613) part_id(number_part),
     +            (part_pos(i,number_part),i=1,3),
     +            (part_dir(i,number_part),i=1,3),
     +              part_mom(number_part),part_time(number_part)
            do i=1,3
               ydir(i)=part_dir(i,number_part)  !RES
               yver(i)=part_pos(i,number_part)  !RES
            end do
```

```
else
          read(8,end=613,err=613) line !skip over rest
        end if
         n_lines=n_lines-1
       end do
       goto 1
613     continue
        number_part=number_part-1
        print *,'Error in KIN line'
       print *,line
       goto 1
       elseif(major_id.eq.'SIM') then
        if(minor_id(1:2).eq.'LS') then
          lsu_simulation=.true.
       else
          lsu_simulation=.false.
       end if
       goto 1            !this line can only be one long
       elseif(major_id.eq.'TUB') then
       tdata_active=.true.
        read(line(ptr1:80),*,err=819,end=819) n_lines
        n_lines=n_lines-1
       n_hits=0
       write(10,100)n_lines
100     format(1x,i2)
       do while(n_lines.ge.1)
          read(8,*,end=813,err=814) tube_number,time,pulse_height
        if((tube_number.gt.0).and.
   &          (tube_number.lt.N_PMTS_MAX))then
           if(pmt_type(tube_number).eq.1)then !ID pmts only RES
            n_hits=n_hits+1
            if (pulse_height.le.0) then
            charge = 0.01
           else
             charge = pulse_height
```

```
            endif
               pmt_dat(1,n_hits)=pmt_position(1,tube_number)
               pmt_dat(2,n_hits)=pmt_position(2,tube_number)
               pmt_dat(3,n_hits)=pmt_position(3,tube_number)
              pmt_dat(4,n_hits)=time * 30.
              pmt_dat(5,n_hits)=charge
              pmt_dat(6,n_hits)=1.
               pmt_dat(7,n_hits)=tube_number
               write(10,200)n_hits,tube_number,
     &     pmt_dat(1,n_hits),pmt_dat(2,n_hits),pmt_dat(3,n_hits)
     &    ,pmt_dat(4,n_hits),time
200     format(1x,i2,1x,i5,f10.2,1x,f10.2,1x,f10.2,1x
     &          ,f10.2,1x,f10.2,1x)
             goto 815
            end if
          else
              print *,'Bad tube number in READ_STXT_EVENT ',
     &           tube_number
           goto 815
          end if
814        continue
           print *,'Error in reading input file'
815          n_lines=n_lines-1
        end do
        tdata_active=.false.
        goto 1
813      continue
         print *,'Unexpected EOF while reading tube data'
        istatus=1
         tdata_active=.false.
        lost=.false.
         evt_active=.false.
        return
819      continue
         print *,'Error in reading input file'
```

```
        return
       elseif(major_id.eq.'TRK')  then
        read(line(ptr1:80),*,err=913,end=913) n_lines
        n_lines=n_lines-1
        do while(n_lines.ge.1)
          read(8,end=913,err=913) line !skip over rest
           n_lines=n_lines-1
       end do
       goto 1
913      continue
        print *,'Error in TRK line'
       print *,line
       goto 1
     else
         print *,'Unrecognized major ID ',major_id
       read(line(ptr1:80),*,err=1013,end=1013) n_lines
       n_lines=n_lines-1
       do while(n_lines.ge.1)
         read(8,end=1013,err=1013) line !skip over rest
          n_lines=n_lines-1
       end do
       goto 1
1013     continue
        print *,'Error in UNREC ID line'
       print *,line
      goto 1
    end if
   else                !did not find special identifier, we are lost
     if(.not.lost) then
       print *,'Lost!, searching from line...'
      print *,line
     lost=.true.
      evt_active=.false.
     tdata_active=.false.
    end if
```

```
      goto 1
    end if
13  continue
     print *,'read error on input'
    istatus=2
    return
99  continue
     print *,'end of file on input'
    istatus=-1
    return
    end
```

VITA

Ebrahim Khosravi received his bachelor of science in electrical engineering from Southern University in May 1985. He obtained his master of science degree in Computer Science from Southern University in December 1993. While working full time at Louisiana State University as research associate with high energy Group, he pursued his doctoral studies in Computer Science.

Ebrahim Khosravi anticipates receiving his doctor of philosophy degree in Computer Science with a minor in physics in May 1998. His current interests are in parallel system, high performance computing, computer architecture.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Ebrahim Khosravi

Major Field: Computer Science

Title of Dissertation: Parallelization and Implementation of Approximate Root Isolation for Nonlinear System by Monte-Carlo

Approved:

Major Professor and Chairman

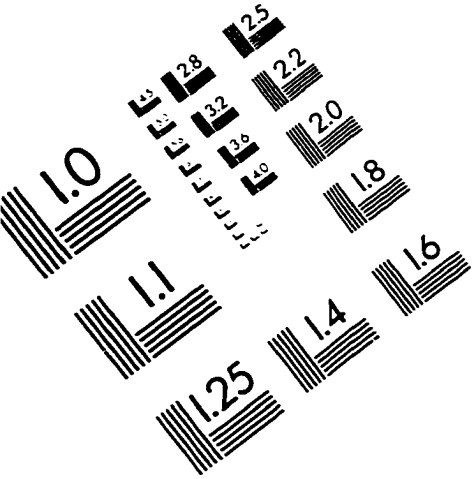
Dean of the Graduate School

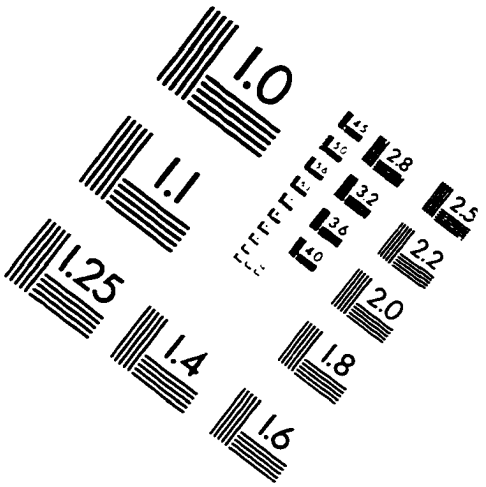
EXAMINING COMMITTEE:

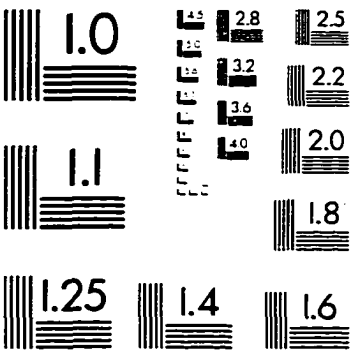
Date of Examination:

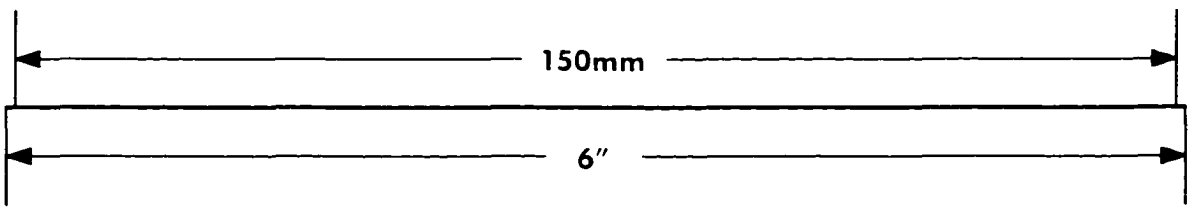
February 4, 1998

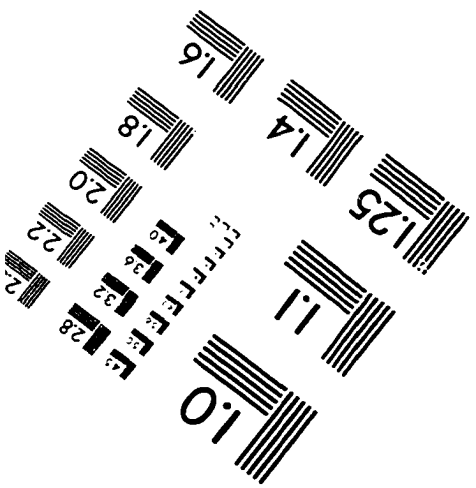
IMAGE EVALUATION
TEST TARGET (QA-3)

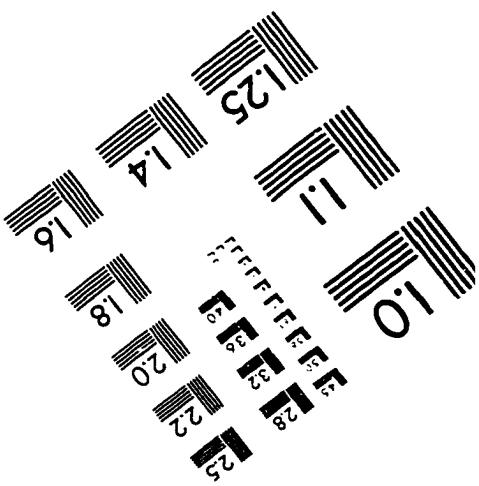












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989